
**Behavioral Fault Modeling
in a
VHDL Synthesis Environment**

A Dissertation

**Presented to
the Faculty of the School of Engineering and Applied Science
University of Virginia**

**In Partial Fulfillment
of the Requirements for the Degree**

Doctor of Philosophy (Electrical Engineering)

by

Ronald J. Hayne

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

May 1999

DTIC QUALITY INSPECTED 4

19990701 057

Approved for public release; distribution is unlimited.

© Copyright by
Ronald J. Hayne
All rights reserved
May 1999

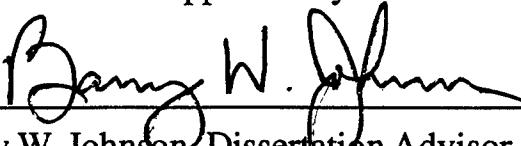
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy (Electrical Engineering)

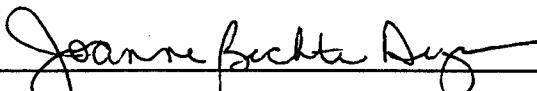


Ronald J. Hayne

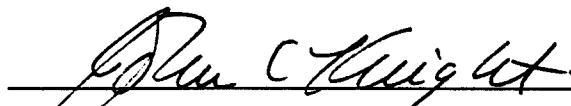
This dissertation has been read and approved by the Examining Committee:



Barry W. Johnson, Dissertation Advisor



Joanne Bechta Dugan, Committee Chair



John C. Knight, Minor Representative



Stephen G. Wilson



Mircea R. Stan

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

May 1999

Abstract

Integrated circuit designs continue to increase in both size and complexity, making fault simulation and testing more difficult and costly. Computer aided design tools and hardware description languages are now commonly used to represent designs at higher levels of abstraction. However, fault simulation and testing of digital circuits have been historically done using fault models at the gate level or below. A design methodology is needed for performing fault simulation throughout the design process, incorporating fault models at higher levels of abstraction. Use of these higher level fault models has the promise of reducing complexity, providing earlier identification of potential problems, and improving integration of fault simulation into the overall design process.

Previous behavioral fault models lack a well defined link to the hardware which they attempt to describe. Though some relationships to possible hardware faults are proposed, there is no detailed analysis to justify these assertions. Approaches based on perturbing language constructs, such as *ADD* to *SUB*, do not accurately reflect underlying hardware faults. In order to compensate for this “big micro-operation problem,” alternate methods such as heuristics are used to supplement test vector sets to increase the equivalent gate level fault coverage.

This dissertation proposes a new set of fault models for VHDL behavioral descriptions of combinational logic circuits. These fault models exploit hardware relationships that exist in a design environment which involves synthesis of behavioral descriptions into gate level circuits. A functional analysis technique is used to evaluate the effects of industry standard *single-stuck-line* (*SSL*) faults on gate level implementations. The generalized functional faults are then abstracted into the behavioral domain by examining their relationship with the higher level language construct.

Test vectors derived from the new behavioral fault models are applied to synthesized gate level realizations of a range of circuits that include typical arithmetic and logic functions. Resulting gate level fault coverage is determined via fault simulation and used as a measure of effectiveness for the new fault models. Because the behavioral faults are derived from a functional analysis of low level faults, they provide improved fault coverage over previous fault models, over a broad range of implementations.

Acknowledgments

I would like to thank the United States Army for sponsoring my continuing education. My advisor, Dr. Barry W. Johnson, deserves a special thanks for his encouragement and support. His technical leadership helped shape many of the ideas embodied in this dissertation. Along with Dr. Johnson, I would also like to thank the other members of my advisory committee, Dr. Joanne Bechta Dugan, Dr. John C. Knight, Dr. Stephen G. Wilson, and Dr. Mircea R. Stan. Their guidance and feedback was invaluable throughout my research. My gratitude is extended to Dr. D. Todd Smith and Todd A. DeLong for serving as a sounding board and sanity check for many of my ideas. Thanks also to the United States Air Force Rome Laboratory, specifically Dr. Warren H. Debany Jr. and James P. Hanna, for additional comments and perspectives. The interactions of Dr. Robert H. Klenke, Dr. Lori M. Kaufman, Dr. James H. Aylor, and Dr. Ronald D. Williams are also greatly appreciated.

I owe a debt of gratitude to the many individuals who have helped me throughout my program. A special thanks is extended to the following individuals: Maximo Salinas, Carl Elks, Dave Garrett, Michael Reynolds, Ryan Baucom, Wes Dungan, and Erik Laurila. Thanks to the EE computer system administrators: Melissa Thrush and Diane Calleson. Thanks also to the secretaries: Lisa Sites, Peggy McCauley, Susan Malone, and Robbie Burton.

My family has provided endless love and support. The biggest thanks to my wife, Marva, for always being there for me. Her encouragement and endless optimism helped me deal with the ups and downs of the past few years. Thanks also to my sons, James and Kyle, for trying to understand all the hours spent in the conquest of this goal. My brother, Tomas, has also been a big source of encouragement. Our weekly outings to the gym helped relieve the stress inherent in such an endeavor. Finally, thanks to my parents, Jim and Audrey, for everything. They brought me up to believe that I could achieve anything I put my mind to.

Biography

Ronald J. Hayne is a Lieutenant Colonel in the U.S. Army with over 18 years of service in a variety of command and staff positions. He received his B.S. in 1980 from the United States Military Academy at West Point and was stationed with the 25th Infantry Division in Hawaii. He received his M.S. in Electrical Engineering in 1987 from the University of Arizona, followed by a teaching assignment with the Department of Electrical Engineering and Computer Science at West Point. LTC Hayne is now a member of the Army Acquisition Corps, responsible for the research and development of new military systems, and has served with the Space and Strategic Defense Command and the Army Research Laboratory. His world travels in the military have taken him to the Republic of Korea, the Republic of the Philippines, Vienna Austria, and Kwajalein Atoll in the Republic of the Marshall Islands.

Publications

Hayne, R.J. and B.W. Johnson, "Behavioral Fault Modeling in a VHDL Synthesis Environment," *Proceedings VLSI Test Symposium*, April 1999.

Kaufman, L.M., R. Gretlein, and R.J. Hayne, "A Quantitative Assessment of the Application of Software Reliability to Reusable Code," *Proceedings Reliability and Maintainability Symposium*, January 1999.

Hayne, R.J., P A. Brown, S.G. Hair and J.E. Oristian, "An Innovative Educational Application of the VHSIC Hardware Description Language," *Proceedings Frontiers in Education Conference*, July 1990.

Table of Contents

Table of Contents	i
List of Figures	vii
List of Tables	xii
List of Acronyms and Abbreviations	xvi
Chapter 1: Introduction	1
1.1 Previous Research	2
1.2 Behavioral Modeling.....	4
1.2.1 VHDL Subset.....	4
1.2.2 Hardware Implementation of VHDL Constructs	6
1.3 Functional Analysis.....	7
1.4 Fault Injection Using WAVES.....	8
1.5 Comprehensive Examples	8
1.6 Contributions and Future Work.....	9
Chapter 2: Previous Research	10
2.1 Fault Models.....	10
2.1.1 Functional Faults	11
2.1.2 Physically-Induced Faults	13
2.1.3 Behavior Faults	15
2.1.4 Model Perturbation	16
2.1.5 Other Research.....	18
2.1.6 Summary	22
2.2 Fault Injection Techniques	22
2.2.1 Instruction Set Architecture	22
2.2.2 MEFISTO.....	23
2.2.3 Hybrid Fault Emulation	23
2.2.4 Summary	24
2.3 Test Generation Techniques.....	24
2.3.1 S-Algorithm	25
2.3.2 B-Algorithm	25
2.3.3 Other Research.....	26
2.4 Conclusions	28
Chapter 3: A New Control Fault Model	30
3.1 IF-THEN-ELSE	30
3.1.1 Synthesis of a Simple Example.....	30
3.1.2 Functional Analysis.....	31
3.1.3 Alternate Implementation	33
3.1.4 Generalized Functional Fault Model.....	35

3.1.5	Development of a Behavioral Fault Model.....	35
3.1.6	Evaluation of the New Behavioral Fault Model	37
3.1.7	Comparison with Previous Behavioral Fault Models	38
3.1.8	Expansion of the Fault Model.....	40
3.1.8.1	Functional Analysis	41
3.1.8.2	Generalized Functional Fault Model	44
3.1.8.3	Behavioral Fault Model	45
3.1.8.4	Evaluation of the Behavioral Fault Model	46
3.1.9	Summary	47
3.2	CASE.....	48
3.2.1	Application of the Control Fault Model.....	49
3.2.2	Evaluation of the Fault Model	51
3.2.3	Comparison with Previous Behavioral Fault Models	53
3.3	Conclusions	55
Chapter 4: Relational Operators.....		56
4.1	Greater Than (GT).....	56
4.1.1	Generalized Functional Faults	57
4.1.2	Classification of Functional Faults	58
4.1.3	Behavioral Fault Model	60
4.1.4	Adapting the Model for GE, LT, and LE.....	62
4.1.5	Summary	63
4.2	Threshold Detection	63
4.2.1	Greater Than Signed Threshold	63
4.2.2	A Quick Example	64
4.3	Equal (EQ).....	66
4.3.1	Functional Faults	66
4.3.2	Behavioral Fault Model	67
4.4	Comparison with Previous Fault Models	68
4.5	Application of the New Fault Models	69
4.5.1	Faults on Relational Operators.....	69
4.5.2	Control Faults.....	71
4.5.3	Final Behavioral Test Vector Set.....	73
4.6	Evaluation of Behavioral Test Vectors	74
4.6.1	Gate Level Realizations	75
4.6.2	Expansion of the Data Path	77
4.6.3	Expansion of the Control Signals.....	79
4.7	Conclusions	80
Chapter 5: Arithmetic Operators.....		82
5.1	Addition.....	82
5.1.1	Ripple Carry Adder	83
5.1.1.1	Functional Testing	84
5.1.1.2	Scalability	86
5.1.1.3	Behavioral Fault Model	86

5.1.1.4	Evaluation of the Behavioral Test Vectors	88
5.1.1.5	Carry-in and Carry-out	89
5.1.2	Carry Look-Ahead Adder	91
5.1.2.1	Functional Testing	92
5.1.2.2	Application of the Behavioral Test Vectors	95
5.1.2.3	Scalability	95
5.1.2.4	Optimization of CLA Behavioral Faults	99
5.1.3	Summary	100
5.2	Subtraction	101
5.2.1	Direct Subtraction	101
5.2.1.1	Functional Testing	101
5.2.1.2	Application of the Behavioral Test Vectors	102
5.2.2	Subtraction Using Addition Circuitry	103
5.2.3	Summary	104
5.3	Constants as Operands	104
5.3.1	Functional Testing	105
5.3.2	Generalized Behavioral Fault Model	105
5.4	Comparison with Previous Fault Models	106
5.5	Conclusions	107
Chapter 6: Other Operators		109
6.1	Logical Operators	109
6.1.1	AND/OR	110
6.1.1.1	Functional Faults	110
6.1.1.2	Complex Expressions	111
6.1.1.3	Scalability	113
6.1.1.4	Behavioral Fault Model	113
6.1.1.5	Application of the New Fault Models	114
6.1.2	XOR	118
6.1.2.1	Functional Faults	118
6.1.2.2	Optimized Test Generation	119
6.1.2.3	Evaluation of the Generalized Test Vectors	122
6.1.3	Comparison with Previous Fault Models	123
6.2	Unary Operators	124
6.2.1	Absolute Value	124
6.2.2	Negation	124
6.2.3	Generalized Functional Faults	125
6.2.4	Behavioral Fault Model	126
6.2.5	Evaluation of Behavioral Test Vectors	126
6.3	Conclusions	128
Chapter 7: Other Programming Constructs		129
7.1	Loops	129
7.1.1	A Simple Example	129
7.1.2	Comparison with Previous Fault Models	134

7.2	Functions and Procedures.....	134
7.2.1	Example ADD4fn	135
7.2.2	Example ADD4pr	136
7.3	Conclusions	137
Chapter 8: Comprehensive Examples.....		138
8.1	Arithmetic Logic Unit	138
8.1.1	Example ALU4wc.....	138
8.1.1.1	Faults on Logical Operators	141
8.1.1.2	Faults on Arithmetic Operators	142
8.1.1.3	Control Faults	144
8.1.1.4	Final Behavioral Test Vector Set	149
8.1.2	Evaluation of the Behavioral Test Vectors	152
8.1.3	Expansion of the Data Path	154
8.1.4	Summary	155
8.2	Error Correcting Circuit	156
8.2.1	Example HAMMING4	156
8.2.1.1	Faults on XOR-only Expressions	156
8.2.1.2	Faults on Other Logical Expressions.....	159
8.2.1.3	Final Behavioral Test Vector Set	162
8.2.2	Evaluation of the Behavioral Test Vectors	163
8.2.3	Expansion of the Data Path	163
8.2.4	Summary	166
8.3	Conclusions	166
Chapter 9: Conclusions and Future Work		167
9.1	Research Contributions	167
9.1.1	Generalized Functional Faults	167
9.1.2	New Behavioral Fault Models	168
9.1.3	Gate Level Fault Coverage of Behavioral Test Vectors	168
9.1.4	Behavioral Test Generation	169
9.1.5	Behavioral Fault Simulation	169
9.2	Future Work	170
9.2.1	Expansion of Behavioral Fault Models.....	170
9.2.2	Tool Development.....	171
9.2.3	Higher Levels of Abstraction	171
9.3	Concluding Remarks	172
References		173
Appendix A: Additional Examples		180
A.1	Array Indexing	180
A.2	Generalization of the Control Fault Model	183
A.3	Signed Comparison	185
A.4	Unsigned Threshold	188

A.5	Adder/Subtractor	189
A.5.1	Faults on Arithmetic Operators	190
A.5.2	Control Faults	190
A.5.3	Evaluation of the Behavioral Test Vectors	191
A.5.4	CLA Implementation	191
A.6	Arithmetic with Constants	193
A.6.1	Example PLUS25	194
A.6.2	Example MINUS25	195
A.7	XOR4	195
A.7.1	Parse Tree Test Vectors	196
A.7.2	Evaluation of Behavioral Test Vectors	197
A.7.3	Optimized Test Vectors	198
A.7.4	Evaluation of Optimized Test Vectors	199
Appendix B: Fault Experiment Results		200
Appendix C: VHDL Behavioral Descriptions		205
C.1	Variables and Signals	205
C.2	Expressions	206
C.3	If Statement	206
C.4	Case Statement	207
C.5	Loop Statements	208
C.6	Process Statement	209
C.7	Procedures and Functions	210
Appendix D: VHDL Synthesis		211
D.1	Level-0	211
D.2	Mentor Graphics	213
D.3	Synopsys	214
D.4	IEEE Draft Standard	216
Appendix E: Hardware Implementation of VHDL Constructs		218
E.1	Structured Logic Design	218
E.2	Mentor Graphics	224
Appendix F: VHDL Source Code		228
F.1	CASE1	229
F.2	ARRAY4	230
F.3	SHIFT4u	231
F.4	LESS2	232
F.5	EQUAL3	233
F.6	GREATER3	234
F.7	LE5	236
F.8	GE23u	237
F.9	LT12u	238
F.10	GT3n	239

F.11	COMPARE	240
F.12	COMPARE3	242
F.13	COMPARE4	244
F.14	COMPARE34	245
F.15	ADD4	246
F.16	ADD4wc	248
F.17	ADD8	250
F.18	SUB4	252
F.19	ADDSUB4	254
F.20	INC4	256
F.21	INC8	257
F.22	DEC4	258
F.23	ADDINC4	259
F.24	PLUS3	260
F.25	MINUS5	261
F.26	PLUS25	262
F.27	MINUS25	263
F.28	SOP1	264
F.29	SOP4	265
F.30	POS1	266
F.31	GT	267
F.32	XOR4	268
F.33	XOR5	270
F.34	ABS4	272
F.35	ABS8	273
F.36	NEG4	274
F.37	NEG8	275
F.38	ALU4wc	276
F.39	ALU8wc	278
F.40	HAMMING4	280
F.41	HAMMING8	282

List of Figures

Chapter 1

Figure 1-1	Design guidelines.....	5
------------	------------------------	---

Chapter 2

Figure 2-1	Three-input majority circuit M	14
------------	--	----

Chapter 3

Figure 3-1	Behavioral description for example IF1.	30
Figure 3-2	Synthesized hardware for example IF1.....	31
Figure 3-3	SOP gate level circuit for MUX21.....	31
Figure 3-4	Karnaugh map for MUX.....	33
Figure 3-5	POS gate level MUX.	34
Figure 3-6	Control fault model for if-then-else.	37
Figure 3-7	Behavioral description for example IF2.	40
Figure 3-8	Synthesized hardware for example IF2.....	41
Figure 3-9	Gate level circuit for example IF2.	41
Figure 3-10	Control fault model for expanded if-then-else.....	45
Figure 3-11	Equivalent if-then-else and case statements.	49
Figure 3-12	Logical adjacencies among clauses.	50
Figure 3-13	WHEN-CORRUPT (AND) faults for example CASE1.....	50
Figure 3-14	Synthesized Structure1 for example CASE1.....	52
Figure 3-15	WAVES test vectors for example CASE1.....	52
Figure 3-16	Synthesized Structure2 for example CASE1.....	53
Figure 3-17	Fault coverage for Structure2 of example CASE1.	53

Chapter 4

Figure 4-1	Karnaugh map for 2-bit GT function.	57
Figure 4-2	Fault classes for 2-bit GT function.	59
Figure 4-3	Fault classes for 3-bit GT function.	60
Figure 4-4	Fault classes for 2-bit GE function.	62
Figure 4-5	Functional test vectors for signed GT threshold.	64
Figure 4-6	Behavioral description for example LE5.	65
Figure 4-7	Behavioral test vectors for example LE5.....	65
Figure 4-8	Synthesized circuit for example LE5.....	65
Figure 4-9	Fault classes for 3-bit EQ function.	66
Figure 4-10	Behavioral description for example COMPARE.....	69
Figure 4-11	Synthesized Structure1 for example COMPARE.	75
Figure 4-12	Alternate set of behavioral test vectors for example COMPARE.	76
Figure 4-13	Synthesized Structure2 for example COMPARE.	76

Figure 4-14	Fault coverage for Structure2 of example COMPARE.....	77
Figure 4-15	Behavioral test vectors for example COMPARE4.....	78
Figure 4-16	Synthesized Structure for example COMPARE4.....	78
Figure 4-17	Behavioral test vectors for example COMPARE3.....	79
Figure 4-18	Synthesized Structure for example COMPARE3.....	80

Chapter 5

Figure 5-1	Ripple carry adder.....	83
Figure 5-2	Behavioral fault model for ripple carry adder.....	87
Figure 5-3	Behavioral description for example ADD4.....	88
Figure 5-4	Synthesized circuit for example ADD4.....	88
Figure 5-5	NOR-only realization of example ADD4.....	89
Figure 5-6	Behavioral description for example ADD4wc.....	90
Figure 5-7	Carry look-ahead adder.....	92
Figure 5-8	CLA implementation of example ADD4.....	95
Figure 5-9	Ripple carry implementation of example ADD8.....	96
Figure 5-10	Fault coverage for ripple carry ADD8.....	96
Figure 5-11	Block diagram of modular CLA adder.....	98
Figure 5-12	Fault coverage for modular CLA adder.....	99
Figure 5-13	Behavioral description for example SUB4.....	102
Figure 5-14	Synthesized circuit for example SUB4.....	103
Figure 5-15	Subtractor implemented with full adders.....	103

Chapter 6

Figure 6-1	Behavioral description for example SOP1.....	111
Figure 6-2	Binary tree representing example SOP1.....	111
Figure 6-3	WAVES test vectors for example SOP4.....	113
Figure 6-4	Behavioral fault model for example SOP1.....	114
Figure 6-5	Behavioral description for example GT.....	114
Figure 6-6	Parse tree for example GT.....	115
Figure 6-7	Synthesized Structure1 for example GT.....	117
Figure 6-8	Synthesized Structure2 for example GT.....	117
Figure 6-9	Fault coverage for Structure2 of example GT.....	118
Figure 6-10	Labeling scheme for Bossen test.....	119
Figure 6-11	Structure Cascade1 for example XOR5.....	120
Figure 6-12	Structure Cascade2 for example XOR5.....	120
Figure 6-13	Modified Bossen test for Cascade2.....	121
Figure 6-14	Generalized Bossen test vectors for example XOR5.....	122
Figure 6-15	Structure4 for example XOR5.....	122
Figure 6-16	Fault coverage for Structure4 of example XOR5.....	123
Figure 6-17	Behavioral fault model for ABS.....	126
Figure 6-18	WAVES test vectors for example ABS8.....	126
Figure 6-19	Synthesized Structure1 of example ABS8.....	127
Figure 6-20	Fault coverage for Structure1 of example ABS8.....	127

Figure 6-21	Synthesized circuit for example NEG8.....	128
-------------	---	-----

Chapter 7

Figure 7-1	Behavioral description for example SHIFT4u.....	129
Figure 7-2	Expanded case statement for example SHIFT4u.....	130
Figure 7-3	WAVES test vectors for example SHIFT4u.....	133
Figure 7-4	Synthesized Structure1 for example SHIFT4u.....	133
Figure 7-5	Fault coverage for Structure1 of example SHIFT4u.....	134
Figure 7-6	Behavioral description for example ADD4fn.....	135
Figure 7-7	Functions for example ADD4fn.....	135
Figure 7-8	Expanded behavioral description for example ADD4fn.....	136
Figure 7-9	Behavioral description for example ADD4pr.....	136
Figure 7-10	Procedure FA for example ADD4pr.....	137

Chapter 8

Figure 8-1	Entity description for example ALU4wc.....	138
Figure 8-2	Architecture description for example ALU4wc.....	139
Figure 8-3	Logical adjacencies among clauses.....	147
Figure 8-4	WAVES test vectors for example ALU4wc.....	151
Figure 8-5	Synthesized Structure1 for example ALU4wc.....	153
Figure 8-6	Fault coverage for Structure1 of example ALU4wc.....	153
Figure 8-7	WAVES test vectors for example ALU8wc.....	154
Figure 8-8	Entity description for example HAMMING4.....	156
Figure 8-9	Architecture description for example HAMMING4.....	157
Figure 8-10	Structure Cascade1 for 4-input XOR-only expression.....	157
Figure 8-11	Structure Cascade2 for 4-input XOR-only expression.....	157
Figure 8-12	Parse tree for expression D(1).....	159
Figure 8-13	WAVES test vectors for example HAMMING4.....	162
Figure 8-14	Synthesized Structure1 for example HAMMING4.....	163
Figure 8-15	Entity description for example HAMMING8.....	164
Figure 8-16	Architecture description for example HAMMING8.....	164
Figure 8-17	WAVES test vectors for example HAMMING8.....	165

Chapter 9

Appendix A

Figure A-1	Behavioral description for example ARRAY4.....	180
Figure A-2	Equivalent case statement for example ARRAY4.....	180
Figure A-3	Logical adjacencies among clauses.....	181
Figure A-4	WAVES test vectors for example ARRAY4.....	182
Figure A-5	Synthesized circuit for example ARRAY4.....	182
Figure A-6	Behavioral description for example CASE2.....	183
Figure A-7	Behavioral description for example GREATER3.....	185

Figure A-8	Fault classes for 3-bit signed GT function.	186
Figure A-9	WAVES test vectors for example GREATER3.	186
Figure A-10	Synthesized Structure1 for example GREATER3.	187
Figure A-11	Synthesized Structure3 for example GREATER3.	188
Figure A-12	Behavioral description for example GE23u.	188
Figure A-13	Behavioral test vectors for example GE23u.	189
Figure A-14	WAVES test vectors for example GE23u.	189
Figure A-15	Synthesized circuit for example GE23u.	189
Figure A-16	Behavioral description for example ADDSUB4.	189
Figure A-17	WAVES test vectors for example ADDSUB4.	190
Figure A-18	Synthesized circuit for example ADDSUB4.	191
Figure A-19	CLA test vectors for example ADDSUB4.	193
Figure A-20	CLA implementation of example ADDSUB4.	193
Figure A-21	Behavioral test vectors for example PLUS25.	194
Figure A-22	Synthesized Structure1 for example PLUS25.	194
Figure A-23	Parse tree for example XOR4.	196
Figure A-24	WAVES test vectors for example XOR4.	197
Figure A-25	Structure1 for example XOR4.	197
Figure A-26	Structure2 for example XOR4.	198
Figure A-27	Structure3 for example XOR4.	198
Figure A-28	Structure4 for example XOR4.	198
Figure A-29	Structure Cascade1 for example XOR4.	199
Figure A-30	Structure Cascade2 for example XOR4.	199

Appendix B

Appendix C

Appendix D

Figure D-1	Process types for Level-0.	213
Figure D-2	Design restrictions for Synopsys	215

Appendix E

Figure E-1	VHDL constructs that map to multiplexer elements.	219
Figure E-2	Hardware implementation for <i>case</i> statement.	220
Figure E-3	Hardware implementation for <i>if</i> statement.	221
Figure E-4	Hardware implementation for vector indexing.	221
Figure E-5	VHDL description for a ripple carry adder.	222
Figure E-6	Hardware implementation of ripple carry adder.	222
Figure E-7	Using functions to represent combinational logic.	223
Figure E-8	Code example for simple <i>if</i> statement.	224
Figure E-9	Code example for <i>if-else</i> statement.	225
Figure E-10	Synthesized hardware for <i>if-else</i> statement.	225
Figure E-11	Code example for <i>case</i> statement.	226

Figure E-12	Synthesized hardware for <i>case</i> statement.	226
Figure E-13	Code example for variable index assignment.	227

Appendix F

List of Tables

Chapter 1

Table 1-1	Levels of detail commonly used in design.....	2
Table 1-2	Predefined VHDL operators.....	5

Chapter 2

Table 2-1	Minimal SIF test set.....	14
Table 2-2	Micro-operation Faults.....	18

Chapter 3

Table 3-1	SSL fault table for SOP MUX.....	32
Table 3-2	Fault reductions.....	32
Table 3-3	Reduced fault table for SOP MUX.....	33
Table 3-4	Reduced fault table for POS MUX.....	34
Table 3-5	Generalized functional fault model.....	35
Table 3-6	SSL faults detected by behavioral test vectors.....	38
Table 3-7	Stuck-data and stuck-control behavioral faults.....	39
Table 3-8	Faults detected by test vector Set 1.....	39
Table 3-9	Fault reductions for example IF2.....	42
Table 3-10	Covering faults for external control line faults.....	42
Table 3-11	Reduced functional faults for example IF2.....	43
Table 3-12	Generalized functional faults for example IF2.....	44
Table 3-13	Behavioral test vectors for example IF2.....	47
Table 3-14	Behavioral test vectors for example CASE1.....	51

Chapter 4

Table 4-1	Reduced functional faults for SOP GT.....	57
Table 4-2	Reduced functional faults for POS GT.....	58
Table 4-3	Functional faults for n-bit GT.....	60
Table 4-4	Behavioral faults for 2-bit GT.....	61
Table 4-5	Functional faults for n-bit GE.....	62
Table 4-6	Behavioral faults for 2-bit GE.....	63
Table 4-7	Functional faults for n-bit EQ.....	67
Table 4-8	Behavioral faults for 2-bit EQ.....	67
Table 4-9	Micro-operation Faults.....	68
Table 4-10	Test vectors for behavioral faults for A > B.....	70
Table 4-11	Test vectors for behavioral faults for A < B.....	71
Table 4-12	Test vectors for THEN-CORRUPT faults.....	72
Table 4-13	Test vectors for ELSIF-CORRUPT and ELSE-CORRUPT faults.....	73
Table 4-14	Final behavioral test vector set for example COMPARE.....	74

Chapter 5

Table 5-1	Truth table for full adder.....	83
Table 5-2	Phase I functional tests.....	84
Table 5-3	Phase II functional tests.	85
Table 5-4	Phase III functional tests.....	85
Table 5-5	Functional tests for 4-bit ripple carry adder.....	86
Table 5-6	Functional tests for example ADD4wc.....	91
Table 5-7	Carries for 4-bit CLA adder.....	92
Table 5-8	Missing carry faults.....	93
Table 5-9	Extra carry faults.....	94
Table 5-10	Additional behavioral test vectors for CLA adder.....	94
Table 5-11	Behavioral test vectors for extra carry faults.	97
Table 5-12	Behavioral test vectors for missing carry faults.....	98
Table 5-13	Optimized test vectors for missing carry faults.	100
Table 5-14	Optimized test vectors for extra carry faults.....	100
Table 5-15	Truth table for full subtractor.....	101
Table 5-16	Functional tests for 4-bit direct subtraction.	102
Table 5-17	Functional tests for adder by subtraction test vectors.....	104
Table 5-18	Functional tests for 4-bit increment function.....	105
Table 5-19	Functional tests for 6-bit function $Z \leq Y - 5$	106
Table 5-20	Micro-operation Faults.....	107

Chapter 6

Table 6-1	Functional faults for AND operation.....	110
Table 6-2	Functional faults for OR operation.....	110
Table 6-3	Functional test vectors for example SOP1.....	112
Table 6-4	Reduced test vectors for example SOP1.....	113
Table 6-5	Behavioral test vectors for example GT.	116
Table 6-6	Reduced test vectors for example GT.	117
Table 6-7	Generalized functional faults for XOR operation.....	119
Table 6-8	Bossen test vectors for Cascade1.....	120
Table 6-9	Modified Bossen test vectors for Cascade2.	122
Table 6-10	Reduced functional faults for 4-bit ABS.	124
Table 6-11	Reduced functional faults for 4-bit negation.	125
Table 6-12	Generalized functional faults for absolute value and negation.....	125

Chapter 7

Table 7-1	Behavioral faults for example SHIFT4u.....	131
-----------	--	-----

Chapter 8

Table 8-1	Behavioral test vectors for OR operator.	141
Table 8-2	Behavioral test vectors for remaining Logical Functions.	142

Table 8-3	Behavioral tests for 4-bit ADD with carry.....	143
Table 8-4	Possible test vectors for Arithmetic Functions.....	143
Table 8-5	Control faults for Logical Functions.....	145
Table 8-6	Control faults for Arithmetic Functions.....	146
Table 8-7	THEN-CORRUPT control faults.....	148
Table 8-8	ELSE-CORRUPT control faults.....	149
Table 8-9	Final behavioral test vectors for Logical Functions.....	150
Table 8-10	Final behavioral test vectors for Arithmetic Functions.....	151
Table 8-11	ALU fault experiment results.....	155
Table 8-12	Optimized test vectors for 4-input XOR-only expression.....	158
Table 8-13	Optimized test vectors for expression S(1).....	158
Table 8-14	Optimized test vectors for expression S(2).....	158
Table 8-15	Optimized test vectors for expression S(3).....	159
Table 8-16	Reduced test vector set for XOR-only expressions.....	159
Table 8-17	Behavioral test vectors for expression D(1).....	160
Table 8-18	Reduced test vectors for expression D(1).....	160
Table 8-19	Reduced test vectors for expression D(2).....	161
Table 8-20	Reduced test vectors for expression D(3).....	161
Table 8-21	Reduced test vectors for expression D(4).....	161
Table 8-22	Test vectors for logical expressions D(1) through D(4).....	162
Table 8-23	HAMMING fault experiments.....	166

Chapter 9

Appendix A

Table A-1	Behavioral faults for example ARRAY4.	181
Table A-2	Behavioral faults and corresponding test vectors for example CASE2. .	184
Table A-3	Coverage for control faults.	191
Table A-4	Missing carry faults for subtraction.	192
Table A-5	Extra carry faults for subtraction.	192
Table A-6	Functional tests for example MINUS25.	195
Table A-7	Behavioral test vectors for example XOR4.	196
Table A-8	Optimized test vectors for example XOR4.	199

Appendix B

Table B-1	Control fault experiments.	200
Table B-2	Relational operator fault experiments.	200
Table B-3	Arithmetic operator fault experiments.	201
Table B-4	Other operator fault experiments.	202
Table B-5	Comprehensive fault experiments.	204

Appendix C

Table C-1	Predefined VHDL operators.	206
-----------	---------------------------------	-----

Appendix D**Appendix E**

Table E-1 Truth table for process MUX2. 220

Appendix F

List of Acronyms and Abbreviations

ABS	Absolute Value
ALU	Arithmetic Logic Unit
ATPG	Automated Test Pattern Generation
BTG	Behavioral Test Generator
CLA	Carry Look-Ahead
CSIS	Center for Semicustom Integrated Systems (University of Virginia)
DEC	Decrement
EQ	Equal
FA	Full Adder
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FS	Full Subtractor
GE	Greater than or Equal to
GT	Greater Than
HA	Half Adder
HS	Half Subtractor
IEEE	Institute of Electrical and Electronics Engineers
INC	Increment
ISA	Instruction Set Architecture
LE	Less than or Equal to
LFSR	Linear Feedback Shift Register
LT	Less Than
MEFISTO	Multi-level Error/Fault Injection Simulation Tool
MIL-STD	Military-Standard
MOD	Modulus
MUX	Multiplexer
NE	Not Equal
POS	Product-of-Sums
RAM	Random Access Memory

REM	Remainder
ROM	Read Only Memory
RTL	Register Transfer Level
SIF	SSL-Induced Fault
SOP	Sum-of-Products
SSL	Single-Stuck-Line
SUB	Subtraction
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
WAVES	Waveform and Vector Exchange
XOR	Exclusive Or

Chapter 1

Introduction

Integrated circuit designs continue to increase in both size and complexity. Fault simulation and testing of these designs is, thus, becoming more difficult and costly. Designers now commonly make use of computer aided design tools and hardware description languages, such as VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language), to represent their designs at higher levels of abstraction. However, fault simulation and testing of digital circuits for manufacturing faults have been historically done using fault models at the gate level or below. Use of these lower level fault models adds complexity and delays these efforts to later in the design cycle.

There is a need to develop a design methodology for performing fault simulation throughout the design process, at many levels of abstraction. It is, therefore, desirable to develop fault models at higher levels of abstraction, based on functional or behavioral descriptions. Working with behavioral fault models will also allow fault simulation to be performed earlier in the design scheme, without details of the gate level implementation. In fact, depending on the source of the component, a gate level description may never be available. Thus, fault simulation and testing based on these higher level fault models have the promise of being less complex, providing earlier identification of potential problems, and improving integration into the overall design process.

This dissertation develops a new set of fault models for VHDL behavioral descriptions of combinational logic circuits. The fault models exploit hardware relationships that exist in a design environment which involves synthesis of behavioral descriptions into gate level circuits. The focus is on relating high level language constructs to lower level hardware faults, as opposed to perturbing the language elements as if they were simply software. A functional analysis technique is used to evaluate the effects of industry standard *single-stuck-line* (SSL) faults on gate level implementations. The generalized functional faults are then abstracted into the behavioral domain by examining their relationship with the higher level language construct. The resulting behavioral fault models are, thus, more strongly linked to underlying hardware faults than those developed by previous research.

As part of this research, test vectors derived from the new behavioral fault models are applied to synthesized gate level realizations of a range of circuits that include typical arithmetic and logic functions. Resulting gate level SSL fault coverage is determined via fault simulation and used as a measure of effectiveness for the new fault models. Because the behavioral faults are derived from a functional analysis of low level faults, they provide improved fault coverage over previous fault models, over a broad range of implementations.

1.1 Previous Research

Recent research efforts have attempted to develop fault models at higher levels of abstraction, which accurately represent faults which occur at lower levels. The models of interest in this research move up the design hierarchy, shown in Table 1-1 [10], beyond the gate level to the register or chip level. Functional fault models are based on the input/output relationship of higher level primitives which may incorporate a large number of gates. Behavioral fault models are based on procedural descriptions of the circuits desired function. Many models reference the register transfer level which describes procedural data flow among functional primitives.

Level of Detail	Behavioral Domain Representation	Structural Domain Primitives
System	Performance specification	Computer, disk, unit, radar
Chip	Algorithm	Microprocessor, RAM, ROM, UART
Register	Data flow	Register, ALU, COUNTER, MUX
Gate	Boolean equations	AND, OR, XOR, FF
Circuit	Differential equations	Transistor, R, L, C
Layout/silicon	None	Geometrical Shapes

Table 1-1 Levels of detail commonly used in design.

The majority of functional and behavioral modeling efforts can be traced to four prominent researchers: Jacob Abraham, James Armstrong, Sumit Ghosh, and John Hayes. Chapter 2 of this dissertation examines the models developed by groups including each of these researchers. Modeling at many levels of abstraction is discussed by Abraham et al.

[1][2][17][65][66], but the main focus of this research concerns the functional fault models. Armstrong and his collaborators [6][7][8][9][22][28][69] have developed an ever evolving series of functional and behavioral fault models and have implemented test generation algorithms using these models. Ghosh and Chakraborty [18][26][27] have proposed a set of fault models which are based on the failure modes of the language constructs of a generic hardware description language. Finally, Hayes et al. [29][31][32] [33][34][63] has worked extensively on fault models for digital circuits, including descriptions of functional fault models leading to a new generic class called *induced* faults.

The survey of the state of the art in high level fault modeling clearly indicates that there is no widely accepted solution to the problem. Modeling techniques range from the functional analysis employed by Abraham and Hayes, to the procedural data flow of behavioral descriptions used by Ghosh and Armstrong. Previous behavioral fault models lack a well defined link to the hardware which they attempt to describe. Though some relationships to possible hardware faults are proposed, there is no detailed analysis to justify these assertions. Approaches based on perturbing language constructs, such as *ADD* to *SUB*, do not accurately reflect underlying hardware faults. In order to compensate for this “big micro-operation problem,” alternate methods such as heuristics are used to supplement test vector sets to increase the equivalent gate level fault coverage.

Though previous research provides no clear cut solution to modeling faults at higher levels of abstraction, valuable insights are gained by the examination of each of these techniques. Certain key concepts from past efforts have immediate applicability here, notably *functional equivalence* and *fault dominance*. Further, the behavioral fault models developed in this dissertation only affect the *activation* step of the test generation process. Hence, the high level algorithms developed to handle the computationally intensive tasks of fault *propagation* and *justification* still remain valid. Integration of new fault models with an existing behavioral test generation algorithm such as the B-algorithm [21][22] can be of mutual benefit. Such algorithms already address problems such as reconvergent fanout, while use of more complex fault models can eliminate the need to supplement test vector sets via heuristics.

1.2 Behavioral Modeling

Hardware description languages can be used to model system behavior at higher levels of abstraction than the traditional gate or circuit level. Languages like VHDL make use of sequential statements, much like conventional programming languages, to describe the desired behavior of a circuit. Several constructs such as *if-then-else* and *case* are normally provided to control the order of execution of these sequential statements. Designers can, therefore, use VHDL to develop behavioral models that can be simulated to verify their correct functioning, prior to generating hardware. VHDL behavioral descriptions and language constructs are detailed in Appendix C.

Modern synthesis tools interpret the behavioral VHDL source code as a description of an electronic circuit. Not all language constructs map directly to hardware in a synthesis environment, therefore, a language subset is defined for use with a specific tool. General modeling guidelines are normally provided to ensure that resulting hardware will be consistent with the designer's intent. Combining these guidelines with the VHDL subset, higher level models can be synthesized to create optimized gate level descriptions.

1.2.1 VHDL Subset

The VHDL behavioral models used in this research describe combinational logic circuits based on the *IEEE Draft Standard for VHDL Register Transfer Level Synthesis* [36]. The draft builds on the foundation laid by the European VHDL Synthesis Working Group's *Level-0 VHDL Synthesis Syntax and Semantics* [25] and incorporates constructs common to synthesis tools by Mentor Graphics [68] and Synopsys [64]. Details can be found in Appendix D.

The standard represents a subset of VHDL with corresponding design guidelines meant to ensure consistent synthesis of gate level netlists from behavioral descriptions. The key VHDL language constructs supported for behavioral modeling are listed below:

- 1) *if* statement, *case* statement, *loop* statement (*for* only).
- 2) *procedure*, *function*.
- 3) *constant*, *variable*, *signal*.
- 4) all predefined VHDL *operators* shown in Table 1-2.

Relational operators like *greater than* ($>$) and *not equal* (\neq) compare like types and return a Boolean result. The *concatenation operator* ($\&$) combines one-dimensional arrays to form a new array with the contents of the right *operand* following the contents of the left *operand*. Finally, the *modulus (MOD)*, *remainder (REM)*, *exponentiation (**)*, and *absolute value (ABS)* *operators* are only defined for integer types. Definitions for all the VHDL operators can be found in Appendix C.

Type	Operators				
Logical	<i>AND</i>	<i>OR</i>	<i>NAND</i>	<i>NOR</i>	<i>XOR</i>
Relational	=	\neq	<	\leq	$>$
Adding	+	-	$\&$		
Unary (sign)	+	-			
Multiplying	*	/	<i>MOD</i>	<i>REM</i>	
Miscellaneous	**	<i>ABS</i>	<i>NOT</i>		

Table 1-2 Predefined VHDL operators.

Design guidelines for modeling combinational logic circuits, summarized in Figure 1-1, are extracted from those defined by Level-0 and Synopsys. Though the guidelines for

-
- 1) Processes which model pure combinational logic must contain a sensitivity list including all the *signals* which are read into the process. All *signals* and *variables* must be assigned in all the conditional branches.
 - 2) Integer types are automatically converted to bit vectors whose width is as small as possible to accommodate all possible values of the type's range, either in unsigned binary for nonnegative ranges or in 2's complement form for ranges that include negative numbers.
 - 3) The *arithmetic operators* "+" and "-" are predefined for all integer operands.
 - 4) *Multiplying operators* ("*", "/", *mod*, and *rem*) are predefined for all integer types with the following restrictions:
 - a) The right *operand* shall be a computable power of 2.
 - b) Neither *operand* shall be negative.
-

Figure 1-1 Design guidelines.

multiplying operators may seem overly restrictive, they still provide adequate flexibility for designing at the register transfer level and may yet be expanded in subsequent revisions to the standard.

Use of this subset is meant to enhance the portability of VHDL designs across synthesis tools conforming to the standard. Hence, it is used here as the basis for defining higher level fault models which have a closer relationship to resulting synthesized hardware. Behavioral fault models for each of these VHDL constructs are developed in Chapter 3 through Chapter 7 of this dissertation.

1.2.2 Hardware Implementation of VHDL Constructs

Several VHDL language constructs lend themselves directly to hardware implementation with common functional modules such as multiplexers. By examining these language to hardware relationships, this research intends to build the foundation on which higher level fault models can be defined, that are more closely related to their underlying gate level counterparts. Some insights are drawn from two resources which directly discuss the relationship between certain VHDL constructs and the ultimate hardware.

One discussion of hardware implementation of VHDL constructs comes from *Structured Logic Design with VHDL* by Armstrong and Gray [10]. In a section titled “Automated Synthesis of VHDL Constructs,” they show the relationship between multiplexers and language constructs that involve selection, like *if* and *case*. Another insight into the relationship between VHDL language constructs and hardware comes from the *VHDL Style Guide for AutoLogic II* by Mentor Graphics [68]. Again, the link is established between the control constructs *if* and *case* and the multiplexer functional building block. These examples reinforce the intuition that a language construct that involves selection leads naturally to a hardware construct that implements selection, the multiplexer. Details can be found in Appendix E.

Armstrong and Gray also discuss program *loops*, *functions*, and *procedures* in relation to hardware [10]. Multiple implementations of a 4-bit adder are used for illustration. Their discussions and examples serve as the basis for the analysis of these programming constructs in Chapter 7 of this dissertation.

1.3 Functional Analysis

Previous research has proposed fault models for behavioral constructs based solely on perturbing the language without a well defined link to the underlying hardware. This dissertation presents new behavioral fault models based on a functional analysis of gate level implementations. By combining VHDL synthesis information with industry standard *SSL* faults, new fault models can be obtained which are more closely linked to the underlying hardware.

A technique has been developed for abstracting *SSL* faults from the gate level into the behavioral domain. First, synthesis information about hardware implementation of VHDL constructs is exploited to obtain a gate level basis for a functional analysis. Next, a reduced set of functional faults, covering all *SSL* gate level faults, is determined with the aid of fault reductions via *functional equivalence* and *fault dominance* [45]. Faults are generalized from various possible implementations to form a set of functional faults not tied to any specific realization. Most importantly, a detailed analysis of the relationship between the generalized set of functional faults and the original VHDL description yields a behavioral fault model for the language construct.

The new fault models developed by this research provide the well defined link to underlying hardware faults that was lacking in previous behavioral fault models. The functional analysis of *SSL* faults takes advantage of VHDL/hardware relationships that exist in a synthesis environment. This analysis of gate level faults adapts the functional techniques employed by Abraham [2] and Hayes [29] with two important extensions. First, functional faults are not tied to a specific implementation, rather they are generalized to be applicable to multiple realizations. Second, the functional faults are further abstracted into the behavioral domain via their relationship with the original VHDL construct being modeled.

Details of the functional analysis technique are presented during the development of the control fault model in Chapter 3. Graphical techniques for examining the relationships between functional faults are first demonstrated with *relational operators* in Chapter 4. Complete functional testing of regular structures of functional building blocks is introduced during the analysis of *arithmetic operators* in Chapter 5. Finally, interactions

among behavioral faults and VHDL constructs are explored throughout the dissertation, but especially in Chapter 7 and Chapter 8.

1.4 Fault Injection Using WAVES

Gate level fault injection experiments were performed throughout this research using a tool developed by DeLong et al. [24]. Test vectors are applied to structural VHDL models using the *IEEE Standard for Waveform and Vector Exchange (WAVES)* [37]. SSL fault simulations are determined using gate level equivalent fault classes according to *MIL-STD 883D* [52]. Test vectors are applied, one at a time, from the input WAVES file. Fault coverage is then evaluated as the ratio of detected faults to total faults and can be plotted versus the test vectors as they are applied.

Fault experiments using behavioral test vectors were conducted during the development of each group of behavioral fault models. The normal requirement for industrial designs is that the set of test vectors provided by the designer achieve at least a 95% SSL gate level fault coverage [27][40][60]. In order to more fully examine the effectiveness of the new fault models, additional examples, not presented in the individual chapters, are provided in Appendix A. Examples have been chosen to represent a broad range of design possibilities. Resulting gate level fault coverage was evaluated to illustrate the effectiveness of the behavioral fault models and is summarized in Appendix B.

1.5 Comprehensive Examples

Two comprehensive example circuits are presented in Chapter 8 to demonstrate the gate level fault coverage of the new behavioral fault models. The first is an arithmetic logic unit (ALU) which performs selected functions on data inputs. The second example is a single error correcting circuit used in fault tolerant applications. Other obvious examples such as a multiplexer or a magnitude comparator do not need to be investigated here due to their detailed analysis as part of the development of the fault models for the *if* statement and *relational operators*.

Application of the behavioral fault models to the comprehensive examples results in a set of test vectors necessary to detect the behavioral faults. These test vector sets are then applied to synthesized gate level implementations. Multiple synthesis tools and target architectures are employed to create a variety of realizations of the behavioral descrip-

tions. AutoLogic II [68] from Mentor Graphics is used with both the autologic/default and gen_lib/default target technologies. The Leonardo [47] synthesis tool from Exemplar Logic is also used to map the behavioral descriptions to a Xilinx field programmable gate array (FPGA) architecture. The resulting gate level fault coverage provides experimental validation of the effectiveness of the behavioral fault models.

1.6 Contributions and Future Work

The main contributions of this dissertation include improved behavioral fault models as well as the techniques for generalizing the effects of low level faults and abstracting them into the behavioral domain. The new fault models are more closely linked to underlying hardware faults than those developed by previous research. Test vectors based on these new behavioral fault models achieve complete *SSL* gate level fault coverage over a broad range of implementations.

The models and techniques presented in this dissertation represent another important step in the development of a design methodology for performing fault simulation throughout the design process. Chapter 9 presents a brief description of directions for future research. These include expansion of the behavioral fault models, tool development for behavioral test generation and behavioral fault simulation, and migration of fault models to even higher levels of abstraction.

Chapter 2

Previous Research

This chapter surveys previous research in three major areas involved in testing circuits based on behavioral descriptions. First, a survey of the current state of the art in fault modeling at higher levels of abstraction provides the basis from which tests for digital circuits can be developed. Recent research efforts have attempted to develop higher level fault models, which accurately represent faults which occur at lower levels. Second, several methods of injecting faults into higher level models are explored. Injecting faults into a behavioral model can be accomplished by manipulating data or signal values within the model or by actually changing or perturbing the model itself. Finally, various techniques for generating behavioral or functional tests provide a means for evaluating the effectiveness of high level modeling efforts. Their results can be compared to more conventional gate level methods to provide a quantitative measure of fault coverage.

2.1 Fault Models

Fault models provide the underlying basis for the development of tests for digital circuits. The models of interest in this dissertation move up the design hierarchy beyond the gate level to the register or chip level. Functional fault models are based on the input/output relationship of higher level primitives which may incorporate a large number of gates. Behavioral fault models are based on procedural descriptions of the circuits desired functioning. Many models reference the register transfer level which describes procedural data flow among functional primitives.

The majority of functional and behavioral modeling efforts can be traced to four prominent researchers: Jacob Abraham, John Hayes, Sumit Ghosh, and James Armstrong. As a foundation for the development and application of more accurate behavioral fault models, this dissertation examines the models developed by groups including each of these researchers. Modeling at many levels of abstraction is discussed by Abraham et al. [1][2] [17][65][66], but the main focus of this dissertation concerns the functional fault models. Hayes et al. [29][31][32][33][34][63] has worked extensively on fault models for digital circuits, including descriptions of functional fault models leading to a new generic class

called *induced* faults. Ghosh and Chakraborty [18][26][27] have proposed a set of fault models which are based on the failure modes of the language constructs of a generic hardware description language. Armstrong and his collaborators [6][7][8][9][22][28][69] have developed an ever evolving series of functional and behavioral fault models and, as will be discussed later, have implemented test generation algorithms using these models.

2.1.1 Functional Faults

In “Fault and Error Models for VLSI,” [2] Abraham and Fuchs provide an extensive review of research efforts aimed at deriving realistic models at higher levels which can accurately represent the faults and errors at lower levels. Of primary interest here are their descriptions of several functional fault models: general fault models for functional blocks, models for small functional modules, and fault models for microprocessors.

Given a combinational function with N inputs, a general fault model assumes that this function can be transformed into any other combinational function of N inputs and, therefore, testing it requires application of all 2^N input combinations. Such exhaustive testing is impractical if N is large, however, testing may be manageable if the function is implemented as an interconnection of subfunctions. The exhaustive general fault model could then be used effectively to test these subfunctions.

Models for several small functional modules provide the building blocks for handling larger functions. A key functional module found in many digital circuits is the decoder. It can be described functionally as having N inputs and 2^N outputs and, under normal operations, exactly one output line is activated corresponding to the input address. A detailed study of all single transistor-level faults by Banerjee [12] resulted in the following functional fault model for a decoder:

- 1) Instead of the correct line, an incorrect line is activated.
- 2) In addition to the correct line, an incorrect line is activated.
- 3) No line is activated.

Though such a description is very simple, it was shown to incorporate all of the physical shorts and opens possible in the transistor-level description.

Further study [1] involved another important building block, the multiplexer. This functional module has N inputs, $\log_2 N$ control signals, and one output. The output is

selected to be one of the inputs as determined by the address on the control lines. Under a fault, it can be shown that the behavior of the multiplexer module can be described in the following functional manner:

- 1) A 0 and a 1 cannot be selected on every input line.
- 2) When selecting some input, another input will be selected instead of, or in addition to, the correct input.

Similar fault models exist for other building blocks of more complex functional units.

Even though microprocessors are quite complex, fairly effective functional fault models have been derived at the register transfer level. Thatte [66] visualizes a microprocessor as a set of functions including register decoding, data transfer, data manipulation, and instruction sequencing. A functional fault model is developed for each of these functions. Improvements to this model made by Brahme [17] are based on the conceptual treatment of instructions as consisting of micro-instructions, which, in turn, are composed of a set of micro-orders. The combined fault model for the microprocessor contains the following:

- 1) Fault Model for the Register Decoding Function:
 - Fault-free
 $f_D(R_i) = R_i$ Register i selected.
 - Faulty
 - $f_D(R_i) = R_j$ Register j selected instead of Register i .
 - $f_D(R_i) = \emptyset$ No register selected.
 - $f_D(R_i) = \{R_i, R_j\}$ Register j selected in addition to Register i .
- 2) Fault Model for the Data Transfer Function:
 - any number of lines can be stuck at 0 or 1.
 - any pair of lines i, j can be coupled.
- 3) Fault Model for the Data Manipulation Function:
 - No specific fault model is presented. (It is assumed that the complete test set for any given ALU can be easily determined.)
- 4) Fault Model for the Instruction Sequencing Function:
 - Under a fault we can have one or more of the following events:
 - One or more microorders can be inactive.
 - Microorders which are normally inactive become active.
 - A set of microinstructions is active in addition to, or instead of, the normal microinstructions.

This approach allows derivation of tests for a microprocessor even if the details of implementation are not known.

With this set of fault models, Abraham and his collaborators have attempted to describe accurately the effects of faults within higher functional modules and thus make complex systems tractable by reducing the number of primitive elements. Though largely based on actual circuit descriptions, care has been taken to make these models as implementation independent as possible by concentrating on the functionality provided by each module.

2.1.2 Physically-Induced Faults

In “Fault Modeling” [34], Hayes also discusses the *general functional (GF)* fault model as one that allows arbitrary changes to a circuit’s truth table (combinational case) or state table (sequential case). The maximum number of states, which can be taken to be one in the combinational case, is assumed to remain constant when faults are present. Detection of *GF* faults requires essentially exhaustive testing procedures and is thus feasible for a moderate number of input lines. Tests for *GF* faults in a sequential circuit are termed checking sequences and tend to be long and difficult to compute. Such tests have, however been applied successfully to the representation of certain types of pattern-sensitive faults in RAMs [33] and to testing simple bit-sliced microprocessors [63].

In more recently published research, Hansen and Hayes [29] present a new high-level fault model called the *physically-induced* fault model. If gate-level *single-stuck-line (SSL)* faults are considered with this model, the authors claim that complete functional fault detection can guarantee complete *SSL* fault detection. The induction concept implies changing the abstraction level at which faults and their effects are considered from a lower to a higher abstraction level. The physical faults of interest include *SSL* faults, *bridging* faults, and *switch-level* faults. The target abstraction level is the functional level, which is loosely defined to correspond to the register-transfer level. Just as the *SSL* fault model is a “natural” gate-level fault model, the authors propose that the *SSL-induced* fault (*SIF*) model is a natural functional-level fault model.

The 3-input majority circuit M , shown in Figure 2-1, is presented as a preliminary example where G , P , and C represent carry-generate, carry-propagate, and carry-in respec-

tively. There are 11 lines producing 22 SSL faults. Analyzing the faulty responses produces 14 different *SIF* functions which can be reduced by *functional equivalence* and *fault dominance* to a minimal set of 6. Table 2-1 lists 6 *SIF* tests that detect all SSL faults.

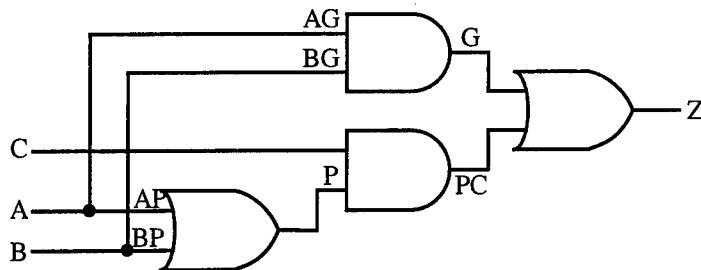


Figure 2-1 Three-input majority circuit M .

SIF	SIF test	Z	C	A	B
A cannot propagate	Propagate with A	1	1	1	0
B cannot propagate	Propagate with B	1	1	0	1
M always propagates	Stop propagate	0	1	0	0
M cannot generate	Generate	1	0	1	1
B always generates	Stop generate with B	0	0	1	0
A always generates	Stop generate with A	0	0	0	1

Table 2-1 Minimal SIF test set.

By examining different realizations of the same majority function, other independent functional faults may be added. Additional tests could, therefore, be required to detect the *SIF* “generate invalidates propagate.” However, given a full set of physical faults, a set of functional faults can be derived, usually without too much difficulty. For example, the work of Shen et al. [62] on inductive fault analysis can be used to supply a comprehensive physical-fault list.

Physically-induced fault techniques have also been applied to several medium-scale examples from the 74X-series. The tests derived for these circuits cover all SSL faults in the standard 74X-series gate-level designs [67], a property that has been verified by complete gate-level fault simulation. Furthermore, the test sets are provably minimal and are generally smaller than those found by an efficient gate-level test generator.

2.1.3 Behavior Faults

In “Behavior-Level Fault Simulation,” [26] Ghosh uses behavioral fault models to represent complex failures in VLSI designs. Faulty values of variables that represent state/timing parameters or a faulty description that is substituted for part of the good description are deliberately introduced into a design that contains no faults. A severe limitation of this method is determining how to select fault models that represent actual failures from the large number that do not. The recommended approach urges designers to use a library of fault models of complex devices that are based on actual failures.

In subsequent research, Ghosh and Chakraborty [18][27] propose fault models based on the failure modes of the language constructs of a generic hardware description language. The programming language C is used to describe hardware with assurances that its language constructs may be extended to other hardware description languages. The fault models presented are relatively complex and attempt to rationalize a link to actual possible hardware faults. Some of the key components of these behavioral fault models include:

- 1) The states of a sequential component may be expressed through variables of integer, Boolean or real types and may fail in one of two modes - the state is permanently held at either V_1 or V_2 where V_1 and V_2 specify the lower and upper extremes of the logical value system.
- 2) A “function call” may exhibit two failure modes where it permanently returns V_1 or V_2 , the lower and upper extremes of the range of the function.
- 3) In the “*for (CC) {E₁}*” clause, the body $\{E_1\}$ may either never be executed or always executed regardless of the condition CC .
- 4) In a “*switch (Id)*” clause, the switch may select either the cases corresponding to the lower or upper extremes of the switch identifier’s value system, all or none of the specified cases.
- 5) The “*if (X) then {E₁} else {E₂}*” construct may fail such that E_1 is always executed and E_2 is never executed, E_1 is never executed and E_2 is always executed, or E_1 and E_2 are executed when X evaluates to false and true respectively.
- 6) The assignment statement “ $X = Y$ ” may fail such that X remains unchanged or assumes the lower (V_1) or upper (V_2) extremes of the value system, or X assumes V_1 or V_2 depending on a probability function.

Though relationships to possible hardware faults are proposed, there is no detailed analysis to justify these assertions. A further shortcoming of these models is the restriction to the lower (V_1) or upper (V_2) extremes of the value system. Multiple bit signals must all be *stuck at 0* or *stuck at 1* rather than allowing for only a *single stuck line*.

In order to evaluate the performance of the proposed behavior fault models, example designs were fault simulated and compared with gate-level fault simulation in the presence of *stuck-at* faults. The example designs included: 16-to-4 multiplexer; 4-, 8-, and 16-bit ALUs with carry lookahead; shift register; synchronous and asynchronous counters; AMD AM2903 bit slice processor; and the control unit of a complex protocol formatter chip. Between 40 and 60 randomly generated test-vector sets were used for gate-level and behavioral fault simulation and the gate-level and behavioral fault coverages were then compared. The researchers found a strong and consistent correlation of the gate-level and behavioral fault coverages with no occurrence of a behavioral fault coverage exceeding 90% while the gate-level fault coverage was less than 85%.

2.1.4 Model Perturbation

Armstrong et al. has developed a series of behavioral fault models based on the concept of model perturbation [28] of designs using hardware description languages, most recently VHDL. In “Behavioral Fault Simulation in VHDL,” [69] Ward and Armstrong define eight behavioral fault classes:

- 1) *Stuck-Then*: represents a failure of the *if-then-else* construct to ever execute the *else* statements.
- 2) *Stuck-Else*: represents a failure of the *if-then-else* construct to ever execute the *then* statements.
- 3) *Assignment Control*: represents a failure of the VHDL assignment operator to assign a new value to a signal.
- 4) *Dead Process*: failure of the statements within a process construct to execute.
- 5) *Dead Clause*: failure of the VHDL *CASE* construct to execute one of the alternative sequences of statements (clauses).
- 6) *Micro-operation*: failure of an operator to perform its intended function. The operator may fail to any other operator in its class.

- 7) *Local Stuck-data*: failure of a signal or variable to have the correct value. The *local stuck-data* fault is restricted to the expression into which it is mapped.
- 8) *Global Stuck-data*: failure of a signal or variable to change value within the device model.

These fault classes have been continually refined by subsequent research, but still they serve as the basis for many current efforts in behavioral test pattern generation.

As a further refinement to their earlier model, Armstrong, Lam, and Ward, [9] subdivide behavioral faults into two broad categories, *control faults* and *micro-operation faults*. *Control faults* perturb the control points that switch between micro-operation sequences. This fault group includes:

- 1) IF: *stuck THEN*, *stuck ELSE* - branching will always occur in one direction, independent of control signal values.
- 2) CASE: *dead clause* - the selected clause does not execute.
- 3) *Assignment* fault - models the effect of a single assignment not taking place.
- 4) *Dead Process* fault.

Micro-operation faults perturb individual micro-operations to another and include:

- 1) AND <-> OR.
- 2) INC <-> DEC.
- 3) ADD <-> SUB.

Some significant problems with this modeling technique include choosing to which micro-operation to perturb [19] and whether any of these faults can actually occur in hardware.

As part of the development of the “B-algorithm: A Behavioral Test Generation Algorithm,” [21][22] Cho and Armstrong developed a new behavioral fault model by applying the concept of *equivalent faults* to the previous model. *Stuck-THEN/stuck-ELSE* faults can be removed from the behavioral fault list if *stuck-at* faults are defined for unnamed signals corresponding to the conditional expressions of the *IF* statement. Likewise, a *micro-operation* fault for a logic operator is detected by a test for a *stuck-at* fault on one of its arguments. Finally, a *dead-clause* fault is equivalent to an *assignment control* fault under the assumption of a single behavioral fault model.

The new behavioral fault model renames *stuck-at* faults to *behavioral stuck-at* faults. *Assignment control* faults are renamed *behavioral stuck-open* faults and *micro-operation*

faults for arithmetic or relational operators are renamed *micro-operation* faults. The reduced model now includes three types of faults:

- 1) *Behavioral Stuck-at (BSA)* Fault - a bit of a signal, virtual signal, a fanout stem, or a fanout branch is permanently *stuck-at* logic 1 or 0.
- 2) *Behavioral Stuck-open (BSO)* Fault - the value of the source expression (right-hand side) of an assignment statement is not correctly transferred to its target.
- 3) *Micro-operation (MOP)* Fault - an arithmetic or a relational operator is faulted to another operator. For example, ADD(A,B) is perturbed to SUB(A,B) and SUB(B,A). A summary of fault-free operators and their corresponding faulty operators is provided in Table 2-2 .

Fault-free Operator	Faulty Operator
ADD	SUB, XOR
SUB	ADD, XOR
BVEQ	BVNEQ
BVNEQ	BVEQ
BVLT	BVGE
BVLE	BVGT

Table 2-2 Micro-operation Faults

Again, perturbing of micro-operations raises doubts concerning the relationship to actual hardware faults. Use of this fault model with the B-algorithm is discussed later along with other test generation techniques.

2.1.5 Other Research

Two other recent research efforts deserve mention here for completeness. The first, by Riesgo and Uceda [60], attempts to define an RTL fault model which they claim is totally oriented to model hardware faults. At the other end of the spectrum, Al Hayek and Robach [4][5] consider behavioral faults as software faults and apply an adaptation of mutation-based testing, originally proposed to test software programs.

In “A Fault Model for VHDL Descriptions at the Register Transfer Level,” [60] Riesgo and Uceda present a fault model based on the VHDL level-0 synthesis subset [25]. Exam-

ple behavioral descriptions can then be directly synthesized and results compared to the corresponding gate-level designs. The fault model is divided into three classes, depending on the type of object affected by the fault:

- 1) Faults on data: the fault model is based on “*stuck-at*” faults. The affected expression will take a constant value and the insertion will be made in a statement where the object is referenced. Examples include:

- bit: *stuck-at-'0'* and *stuck-at-'1'*.
- enumerated: *stuck-at-“all possible values”*.
- integer: each bit of the resulting bus can be *stuck-at'0'* or *stuck-at-'1'*.

The assumed codification is binary for positive numbers and 2's complement for negative numbers.

- 2) Faults on expressions: the fault model is based on “*stuck-at*” faults. The affected expression will take a constant value. Examples include:

- *if_then_else*: the condition of the if statement may be *stuck-at-true* or *stuck-at-false*.
- *case_is_when*: the expression which controls the case statement may be *stuck-at-“all possible values”*
- *for_in_loop*: the index controlling the loop may change its range from the *minimum* to the *maximum+1* and from the *minimum-1* to the *maximum*.

- 3) Faults on statements: the fault model is based on “*dead*” faults. The effect of the fault is that the affected statements are not executed. Examples include:

- *if_then_else*: *dead-then, dead-else*.
- *case_is_when*: *dead-alternative*.
- *for_in_loop*: *dead-loop*.
- procedure call: *dead-call*.
- signal or variable assignment: *dead-assignment*.

Fault insertion is made on the VHDL code by adding code perturbations to the input description. Code perturbations consist of adding, switching, or eliminating code sentences, to model the circuit behavior under a fault condition.

Experiments were conducted on several VHDL descriptions which were then synthesized in order to obtain a comparison with the *stuck-at* fault model at the logic gate level.

The example circuits included 8- and 16-bit ALUs, sequential multiplier, shift register, 16-bit counter, and a reception-transmission unit. Test vectors were randomly generated, then VHDL/RT and logic fault coverages were compared. The fault model presented did not give a precise value of the fault coverage achieved at the lower levels, rather an estimation. For those circuits with a large combinational part, the estimation of fault coverage was “optimistic” and the synthesis options had an influence on the accuracy of the estimation. Highly sequential circuits produced “pessimistic” results due to a large number of faults at the RT level that were very difficult to detect or even undetectable. The achieved results are claimed better than Armstrong and Ghosh due to their correlation coefficients being larger than 90%.

In contrast to other hardware oriented approaches, Al Hayek and Robach [4] propose a mutation-based testing strategy in which VHDL behavioral faults are considered as software faults. The generated test set is used to validate the VHDL description, seen as a software program, against (software) design faults as well as its hardware implementation against hardware faults. A VHDL subset is also used, which allows high level functional description of any combinational or sequential circuit that can be easily synthesized by current tools.

The mutation-based fault model [5] defines a set of mutation operators for use with VHDL behavioral descriptions. Mutation operators include:

- 1) AOR: Arithmetic Operator Replacement.
 - replace “+” with “-”.
- 2) ABS: ABSolute value insertion.
- 3) CR: Constant Replacement
 - integer: increment and decrement by 1.
 - bit-vector: 1’s complement.
 - Boolean: complement.
- 4) CVR: Constant for Variable Replacement.
 - every compatible constant.
- 5) LOR: Logical Operator Replacement.
 - AND, OR, NAND, NOR, XOR
 - replaced by each of the other operators

6) ROR: Relational Operator Replacement.

- <, >, <=, >=, =, /=

- replaced by each of the other operators

7) NOR: No Operation Replacement.

8) VCR: Variable for Constant Replacement.

- every compatible variable.

9) UOI: Unary Operator Insertion.

- each arithmetic expression negated, incremented, and decremented by 1.

- each logical expression complemented.

An automated test environment was built by translating VHDL to FORTRAN and using an existing software testing system.

On a behavioral VHDL description, the test environment delivers a test set and the associated mutation score value that represents the percentage of non-equivalent detected mutants by the test set. In the context of design and test of hardware systems, the mutation score is viewed as a behavioral fault coverage that measures how well the design has been tested. On a set of high-level synthesis benchmarks (Decoder, ALU, Differential equation, Elliptical wave filter,...), experimental results show that on combinational circuits the obtained gate-level fault coverage is about 94% in the average against 99% for the traditional gate-level ATPGs. However, on sequential circuits the mutation-based test is claimed better as it yields 94% in the average of gate-level fault coverage against 85% for the traditional ATPGs, when they are used without any user assistance.

In order to improve the performance of the mutation-based technique, the authors chose to enhance the test set for certain *complex* operators such as AOR. Mutation analysis does not take into account the size of the hardware implementation, because it considers addition/subtraction as a software operation and consequently generates only one test vector. A heuristic is proposed to supplement the original test set with extra vectors to sufficiently test the complex operators. This heuristic consists of generating N new test vectors for each complex operator, where N is the maximum dimension of the input parameters. The necessity to supplement the original test set simply attempts to cover up an underlying deficiency in the mutation-based fault model.

2.1.6 Summary

All of the modeling efforts presented here attempt to develop fault models at higher levels of abstraction, which accurately represent faults which occur at lower levels. This discussion has been meant to provide a survey of the current state of the art in behavioral fault modeling. Later in this chapter, some of these fault models will serve as the basis for several behavioral test generation algorithms. But first, some techniques for injecting faults into behavioral models will be briefly discussed.

2.2 Fault Injection Techniques

Once a fault model has been defined, some method of injecting these faults into a model of the digital circuit must be developed. Injecting faults into a behavioral model can be accomplished by manipulating data or signal values within the model or by actually changing or perturbing the model itself. An example of the signal manipulation technique is provided by the work of DeLong, Johnson, and Profeta in “A Fault Injection Technique for VHDL Behavioral-Level Models.” [23] Modification of behavioral models is demonstrated by Jenn et al. in “Fault Injection into VHDL Models: The MEFISTO Tool.” [39] Finally, Yount and Siewiorek present an approach called hybrid fault emulation in “A Methodology for the Rapid Injection of Transient Hardware Errors.” [71]

2.2.1 Instruction Set Architecture

In order to perform fault injection experiments early in the design cycle, DeLong, Johnson, and Profeta [23] developed a technique to inject faults into a VHDL behavioral model of a system. This technique is demonstrated on an instruction set architecture (ISA) model of an embedded control system. Single or multiple bit faults are injected into internal processor registers, any location in memory, and any range of locations in the memory map. Signal values are corrupted by using a user-defined VHDL data type to communicate with a bus resolution function. When two different sources are trying to update a signal at the same time, the bus resolution function resolves the conflict and assigns the desired value to the signal. This technique allows the designer to inject faults on desired signals in a behavioral description with minimal changes to the existing code. Because the functionality of the design is not changed, the same model can be used to simulate both fault-free and faulty behavior.

2.2.2 MEFISTO

In contrast to the previous technique, the MEFISTO Tool [39] uses two other techniques to inject faults into VHDL models. The first category requires modification of the VHDL model and the second one uses the built-in commands of the simulator. Modification of the VHDL model is accomplished through the addition of components called *saboteurs* and *mutants*. A *saboteur* is a VHDL component that alters the value or timing characteristics of one or several signals when activated. A *mutant* is a component description that replaces another component description. When activated, it imitates the component's behavior in the presence of faults. Both signal and variable manipulations can be used for controlling, i.e., activating and deactivating, *saboteurs* and *mutants*. In this way, the injection of faults can be controlled by built-in commands of the simulator.

The main reason for using the built-in commands of the simulator for fault injection is that this does not require the modification of the VHDL code. However, the applicability of these techniques depends strongly on the command languages of the simulators. The values of either signals or variables may be manipulated by stopping and restarting the simulation. For example, a temporary stuck at fault may be injected by application of the following sequence of pseudo commands:

- 1) SimulateUntil <fault injection time>
- 2) FreezeSignal <signal name> <signal value>
- 3) SimulateFor <fault duration>
- 4) UnFreezeSignal <signal name>
- 5) SimulateFor <observation time>

For a permanent fault, steps 3 and 4 are skipped. Intermittent faults can be injected using a more complex command sequence.

2.2.3 Hybrid Fault Emulation

Yount and Siewiorek [70][71] developed a fault injection methodology for processors based on a register transfer level fault model. The approach, called hybrid fault emulation, uses the actual circuit to perform fault injection. A transient fault occurs during the interval $T = [t_j, t_k]$. The system is allowed to run until some time, t_j , and the state of the machine is captured. A low level model is then used over a limited interval with the cap-

tured state. Reverse fault emulation is used to generate output and set the new state at t_k to match the effects of the desired fault. Since fault simulations are only run over short periods of time, many more fault injection experiments can be run using the same simulator resources. This method is, however, limited to only transient faults and applies to the evaluation of an actual system rather than one still in the design process.

2.2.4 Summary

Though brief, this survey of fault injection techniques provides sufficient insight into methods for manipulating behavioral models. Fault injection can be as simple as starting and stopping the simulation to change desired values or as complicated as developing *mutants* that imitate a component's behavior in the presence of faults. The bus resolution function technique is simply an adaptation of the concept of a *saboteur* that alters the value of a signal in the model. Test designers now have a range of fault models and injection techniques from which to develop behavioral test generation algorithms.

2.3 Test Generation Techniques

Test generation techniques at higher levels of abstraction can be based on either functional or behavioral descriptions and their corresponding fault models. This dissertation investigates research efforts which have produced automated test pattern generation (ATPG) methods and tools to support circuit designers at or near the register transfer level. Results of these test generators can be used to evaluate the effectiveness of the underlying behavioral fault models. Lin and Su [49][50] developed a functional test generation algorithm which uses a register transfer level fault model based on the functional fault work of Abraham [2][17][66]. Armstrong and his collaborators [13][14][15][21][22][46][55][56][57][58] have produced a series of test generators based on their ever evolving behavioral fault models. Several other researchers [20][54] have also developed ATPG algorithms which utilize the fault models of Ward and Armstrong [69]. Finally, Santucci and Giambiasi et al. [59][61] have produced a prototype model of a test pattern generator which uses a fault model claimed equivalent to that of Ghosh and Chakraborty [18][27]. However, since their research focus is on optimization of the test generation algorithm, no results are published that can help evaluate the effectiveness of their fault model.

Most fault-oriented techniques use some form of a three step approach to the test generation process. First, a fault must be *activated* at the desired location in the circuit model. Then, the effect of the fault must be *propagated* to a point where it can be observed and, hence, detected. Finally, the inputs of the model must be determined to *justify* the desired signal values throughout the circuit. Variations of these techniques attempt to utilize the information available in these higher level models to more efficiently accomplish the computationally intensive tasks of fault *propagation* and *justification*.

2.3.1 S-Algorithm

The S-Algorithm [49][50] is based on a register transfer (RT) level fault model similar to the functional fault models discussed by Abraham and Fuchs [2]. The reduced fault set is divided into three groups for ease of fault identification:

- 1) *Register decoding* faults and *operator decoding* faults - regular registers and ALU operators are “global” among RT-statements.
- 2) *Condition* faults, *jump* faults, and *data transfer* faults with constant source registers - they are “local” within an RT-statement.
- 3) *Data transfer* faults with regular source registers.

The overall test generation algorithm also includes a preprocess stage which includes partitioning the system under test and a postprocess stage that evaluates fault coverage.

Major parts of the heuristic test generation algorithm were implemented in IBM Pascal and experimental results have been published for several example circuits. For a hardware multiplier described by 16 RT-statements, the program generated 66 test patterns and claims a 96.4% fault coverage. For the SIMPLE-CALCULATOR, 134 test patterns were generated resulting in a 97.2% fault coverage. No comparisons are provided to evaluate the resulting fault coverages versus a traditional gate level stuck at model.

2.3.2 B-Algorithm

Early test generation algorithms based on VHDL behavioral descriptions included “A Heuristic Chip-Level Test Generation Algorithm” [13] and “The E-Algorithm, an Automatic Test Generation Algorithm for Hardware Description Languages.” [55] Both were strongly influenced by the pioneering work of Levendel and Menon in “Test Generation Algorithms for Computer Hardware Description Languages.” [48] The fault models used

were evolving versions of the model published by Ward and Armstrong [69]. Reported results on medium complexity circuits were in the range of 90% gate level fault coverage.

Further along in the evolution, the Behavioral Test Generator (BTG) [57][58] uses a VHDL subset and the same behavioral fault model [69]. The faulted operation is first *activated* (fault sensitization), then the effect of the fault is *propagated* to the output (fault *propagation*). Examples are provided for *propagation* through behavioral control constructs and *propagation* through data paths. Behavioral fault coverage is then evaluated as the ratio of detected behavioral faults to the total number of behavioral faults.

BTG was tested using a set of 11 logic circuits representing a cross section of generic types of logic. The average equivalent gate level coverage for the experiments on these circuits was 92%. A major drawback was what they called the “big micro-operation problem.” Faults can be applied to micro-operations that represent large blocks for logic. For example $ACUM \leq ADD(A,B,CIN)$ implies a multi-bit adder. It is impossible for a single vector to detect the majority of gate level faults in such a complex structure. O’Neill et al., thus, resorted to heuristics to supplement their test vector set to bring up their equivalent gate level coverage. The necessity to add test vectors to those generated by their algorithm points back to a fundamental deficiency in their underlying fault models.

Continuing efforts by the same research group produced the B-algorithm [21][22]. This test generation algorithm uses the reduced behavioral fault model discussed earlier in this chapter: *behavioral stuck-at* faults, *behavioral stuck-open* faults, and *micro-operation* faults. Again, a three step approach is applied to the test generation process: *activation*, *propagation*, and *justification*. As with BTG, the B-algorithm cannot generate sufficient test vectors for micro-operation faults. In order to raise their equivalent gate level coverage numbers to acceptable levels, an additional $4n-1$ test vectors are generated by a heuristic test generator for each n -bit micro-operation. Even with the stated improvements and the modified fault model, the B-algorithm still suffers from the same underlying deficiencies concerning the modeling of faults in complex operations.

2.3.3 Other Research

In “Analysis of the Gap Between Behavioral and Gate Level Fault Simulation,” [20] Chen and Perumal describe the details of an ATPG system for VHDL behavioral models.

The behavioral fault models based on Ward and Armstrong [69] are expanded and classified into six different categories:

- 1) *Input stuck-at* fault
- 2) *If stuck then* fault
- 3) *If stuck else* fault
- 4) *Assignment statement* fault
- 5) *Dead clause* fault
- 6) *Local stuck data* fault.

The problems associated with micro-operation faults have been avoided by eliminating such constructs from their behavioral models.

For the simple example discussed in their paper, 34 of the 37 non redundant gate level faults are detected by behavioral fault simulation, resulting in a fault efficiency of 92%. The results of a more complicated example of a vending control unit are also presented. A total of 250 test patterns are required to test for 217 faults in the synthesized gate level circuit. Detection of 189 faults results in a fault coverage of 87%. If scan design techniques are applied to the circuit, then a fault coverage of 98% is claimed to be achieved.

The last example to be discussed is the “Behavioral Fault Simulation and ATPG System for VHDL.” [54] For purposes of comparison, VHDL behavioral code is synthesized to gate-level implementations and analyzed. The underlying fault models are again based those proposed by Ward and Armstrong [69], this time including micro-operation faults. A linear feedback shift register (LFSR) algorithm was utilized to generate the test patterns in this research.

Nine circuits were used to evaluate the performance of the system. Actual behavioral fault coverages ranged from 18% to 100%, with sequential descriptions, such as counters and controllers, performing worst and combinational circuits performing best. These results were compared to randomly generated test patterns applied to synthesized gate level circuits. Many of the resulting gate level descriptions proved to be difficult to test with the random test patterns and, therefore, also had low fault coverage numbers. Hence, the authors were able to claim success based on a different figure of merit, relative detection of testable gate-level faults. Using this questionable comparison, Noh et. al. claims an overall result of detecting around 98% of all testable gate level faults. There is dubious

merit in claiming that detecting only 18% of the behavioral faults in a vending machine controller versus 19% in the synthesized gate level circuit implies a 95% success rate.

2.4 Conclusions

This survey of the state of the art in high level fault modeling and test generation clearly indicates that there is no widely accepted solution to the problem. Modeling techniques range from the functional analysis employed by Abraham and Hayes, to the procedural data flow of behavioral descriptions used by Ghosh and Armstrong. Test generation algorithms contained some functional techniques, but mainly relied on the behavioral fault models developed by Armstrong. Examples provided a figure of merit by which to judge their effectiveness, but few common measures could be found. When results did not meet expectations, alternate methods, such as heuristics or testability, were invoked to improve the statistics.

In the functional arena, Abraham has attempted to describe accurately the effects of faults within higher functional modules and thus make complex systems tractable. Though largely based on actual circuit descriptions, care has been taken to make these models as implementation independent as possible. With his *physically-induced* fault techniques, Hayes has been able to derive minimal test sets for several medium-scale examples. Key to this research are the concepts of *functional equivalence*, *fault dominance*, and *compatible fault sets*. Both of these functional methods show promise for application to behavioral faults and hardware description languages.

For behavioral models, though Ghosh proposes relationships to possible hardware faults, there is no detailed analysis to justify these assertions. A further shortcoming of these models is the restriction that multiple bit signals must all be *stuck at 0* or *stuck at 1* rather than allowing for only a *single stuck line*. Similarly, the evolving set of behavioral fault models by Armstrong and their subsequent test generation algorithms seem to move too far away from the hardware which they attempt to describe. In order to compensate for what they call the “big micro-operation problem,” the researchers resorted to heuristics to supplement their test vector set to increase their equivalent gate level coverage. The necessity to add test vectors to those generated by their algorithm points back to a fundamental deficiency in the underlying fault models.

The behavioral fault models developed in this dissertation only affect the *activation* step of the test generation process. Hence, the high level algorithms developed to handle the *propagation* and *justification* steps still remain valid. Integration of new fault models with an existing behavioral test generation algorithm can be of mutual benefit. Such advanced test generation algorithms already address problems such as reconvergent fanout [3], while use of more complex fault models can eliminate the need to supplement test vector sets via heuristics.

Though previous research provides no clear cut solution to modeling faults at higher levels of abstraction, valuable insights have been gained by the examination of each of these techniques. Certain key concepts from past efforts have immediate applicability to further research. Other works can also serve as benchmarks for comparison of future results. Behavioral fault modeling remains an active research area which requires continued exploration.

Chapter 3

A New Control Fault Model

Previous research has proposed fault models for the control constructs *if-then-else* and *case*, such as *stuck-then/stuck-else* and *dead clause*, based solely on perturbing the language without a well defined link to the underlying hardware. This dissertation proposes a new behavioral fault model based on a functional analysis of gate level implementations. By combining VHDL synthesis information with industry standard *single-stuck-line* (SSL) faults, a new control fault model can be obtained which is more closely linked to the underlying hardware.

3.1 IF-THEN-ELSE

The first control construct to be modeled will be *if-then-else*, common to most hardware description languages. In Appendix E Armstrong and Gray identify the link between the *if* statement and the functional building block of a multiplexer. Likewise, Mentor Graphics demonstrates how the selection activity of an *if-then-else* construct is implemented in hardware as a multiplexer. These examples reinforce the underlying intuition that a language construct that involves selection leads naturally to a hardware construct that implements selection, the multiplexer.

3.1.1 Synthesis of a Simple Example

As a first example, an *if* statement is used to select one of two input signals to be assigned to an output signal. The VHDL behavioral description is shown in Figure 3-1.

```
if SEL = '0' then
    Z <= Y0;
else
    Z <= Y1;
end if;
```

Figure 3-1 Behavioral description for example IF1.

The VHDL code was compiled using the Mentor Graphics' Design Architect and then synthesized using AutoLogic II. The resulting circuit shows the expected multiplexer architecture in Figure 3-2.

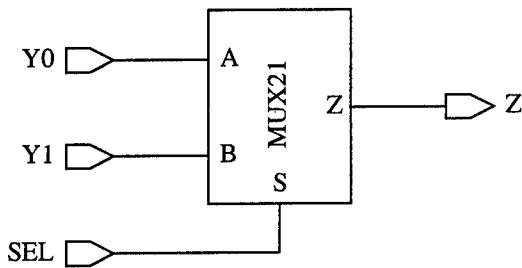


Figure 3-2 Synthesized hardware for example IF1.

The synthesized hardware implementing example IF1 has the two inputs, Y_0 and Y_1 , connected to inputs A and B respectively of a 2-to-1 multiplexer, MUX21. In subsequent discussions, these will be referred to as the *channels* of the multiplexer, *Channel A (CHA)* and *Channel B (CHB)*.

3.1.2 Functional Analysis

In order to perform a functional analysis similar to the method used by Hansen and Hayes [29] in their work on *physically-induced* faults, a gate level design of the multiplexer architecture is needed. To make the resulting models independent of any specific implementation, several different gate level realizations will be examined and compared.

The first gate level multiplexer was obtained by expanding the functional element MUX21 one level lower in the design hierarchy. The resulting gate level circuit is recognized as a *sum-of-products (SOP)* implementation shown in Figure 3-3.

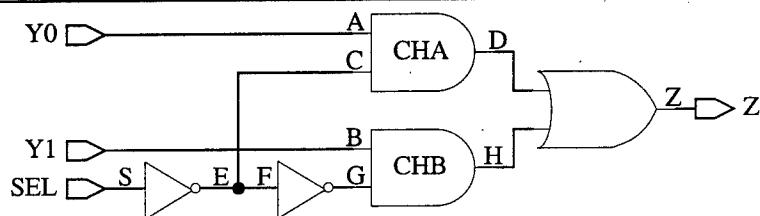


Figure 3-3 SOP gate level circuit for MUX21.

To analyze the gate level circuit, the effect of *single-stuck-line (SSL)* gate level faults will be examined. This analysis will determine a set of functional faults which are induced by the lower level *SSL* faults. Testing for these functional faults will then ensure complete testing for the original gate level faults.

The gate level circuit contains 10 logical lines. Applying the *SSL* fault model where each line can be either *stuck-at-0* or *stuck-at-1*, there are a total of 20 gate level *SSL* faults

in the circuit. The line labeled *A* being *stuck-at-0* and *stuck-at-1* will be indicated as *A-0* and *A-1* respectively. By activating each SSL fault individually in the gate level circuit and evaluating the resulting output response to changing inputs, a fault table is obtained. In Table 3-1 the fault free behavior of the circuit is shown in column *Z*. For simplicity, only outputs due to a SSL fault that differ from the fault free behavior of the circuit are shown.

SEL	Y1	Y0	Z	A-0	A-1	B-0	B-1	C-0	C-1	D-0	D-1	E-0	E-1	F-0	F-1	G-0	G-1	H-0	H-1	S-0	S-1	Z-0	Z-1	
0	0	0	0		1					1									1				1	
0	0	1	1	0				0		0	0									0	0			
0	1	0	0		1					1	1		1			1		1	1	1	1		1	
0	1	1	1	0				0		0											0			
1	0	0	0				1			1									1				1	
1	0	1	0					1		1	1		1						1	1			1	
1	1	0	1			0						0		0	0		0	0	0	0	0	0		
1	1	1	1			0								0	0		0					0		

Table 3-1 SSL fault table for SOP MUX.

Faults which cause the same faulty output can be considered *functionally equivalent* [45]. Such faults can be combined in the fault table, since there is no way to distinguish between these faults by observing the circuit's output behavior. To further reduce the functional faults, consider the concept of *dominance* of one fault over another. The fault (column) *F1* is said to *dominate* the fault (column) *F2* if *F1* has a faulty output in at least every row in which *F2* has a faulty output [45]. The dominating fault (column) *F1* may be removed from the fault table, since any test which detects fault *F2* will also detect fault *F1*. Fault reductions due to *functional equivalence* and *dominance* are shown in Table 3-2.

Faults	Remarks	Faults	Remarks
A-1, E-0, H-1, S-1, Z-1	Dominate G-1	C-0, D-0	Equivalent to A-0
B-1, D-1, E-1, S-0	Dominate C-1	F-0	Equivalent to G-1
Z-0	Dominate B-0	F-1, G-0, H-0	Equivalent to B-0

Table 3-2 Fault reductions.

Only four functional faults remain after applying the indicated fault reductions. Testing for this reduced set of faults will ensure complete coverage of all 20 original *SSL* gate level faults. The reduced fault table and appropriate test vectors are shown in Table 3-3. The two test vectors labeled mandatory are the only ones that cover a specific fault. For the other faults two options are available; selecting one test vector from each group will provide coverage for the faults in question.

Test Vector	SEL	Y1	Y0	Z	A-0	G-1	B-0	C-1
	0	0	0	0				
Option 1	0	0	1	1	0			
Mandatory	0	1	0	0		1		
Option 1	0	1	1	1	0			
	1	0	0	0				
Mandatory	1	0	1	0				1
Option 2	1	1	0	1			0	
Option 2	1	1	1	1			0	

Table 3-3 Reduced fault table for SOP MUX.

3.1.3 Alternate Implementation

In order to investigate an alternate gate level implementation of the multiplexer architecture, consider the Karnaugh map for the output function Z , shown in Figure 3-4. The groupings of minterms (1's), indicated by the dashed lines, produce an *SOP* representation consistent with the gate level circuit previously analyzed from Figure 3-3.

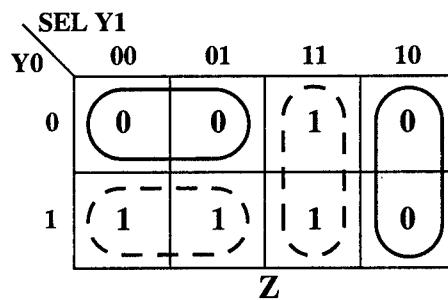


Figure 3-4 Karnaugh map for MUX.

To obtain a *product-of-sums* (*POS*) implementation of the multiplexer, the maxterms (0's) are grouped, as indicated with the solid lines. The resulting function for the output is: $Z = (SEL + Y_0) \cdot (\overline{SEL} + Y_1)$. A gate level realization is shown in Figure 3-5.

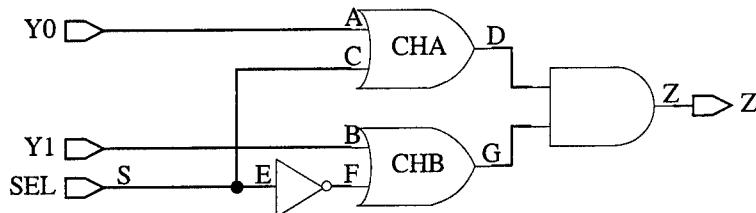


Figure 3-5 POS gate level MUX.

A functional analysis can now be performed on the *SSL* gate level faults for this circuit. After appropriate reductions for *functional equivalence* and *fault dominance*, the resulting fault table is shown in Table 3-4.

Test Vector	SEL	Y1	Y0	Z	A-1	F-0	B-1	C-0
Option 1	0	0	0	0	1			
Mandatory	0	0	1	1		0		
Option 1	0	1	0	0	1			
	0	1	1	1				
Option 2	1	0	0	0			1	
Option 2	1	0	1	0			1	
Mandatory	1	1	0	1				0
	1	1	1	1				

Table 3-4 Reduced fault table for POS MUX.

It should be noted here that a NAND-NAND realization of the *SOP* circuit and a NOR-NOR realization of the *POS* circuit produce the same reduced fault tables shown in Table 3-3 and Table 3-4, respectively. The reduced set of functional faults only affect the controlling inputs to the multiplexer channels, which remain unchanged due to conversions to NAND-only or NOR-only circuits using DeMorgan's theorem [41].

3.1.4 Generalized Functional Fault Model

Comparison of the reduced fault tables for both the *SOP* and *POS* implementations of the multiplexer can now yield a generalized functional fault model, not tied to a specific realization. There are no contradictions among the test vectors indicated in the two fault tables. Selection of the mandatory test vectors from each table provides complete coverage of the functional faults from both tables.

A generalized functional fault model is, therefore, presented in Table 3-5. The functional faults have been renamed *SOP_A*, *SOP_B*, *POS_A*, and *POS_B* to indicate the origin of their mandatory test vector and the channel of the multiplexer which they corrupt. Testing based on the indicated vectors should provide complete coverage of gate level *SSL* faults for multiple multiplexer implementations.

Test Vector	SEL	Y1	Y0	Z	SOP_A	POS_A	SOP_B	POS_B
	0	0	0	0				
Mandatory	0	0	1	1		0		
Mandatory	0	1	0	0	1			
	0	1	1	1				
	1	0	0	0				
Mandatory	1	0	1	0			1	
Mandatory	1	1	0	1				0
	1	1	1	1				

Table 3-5 Generalized functional fault model.

3.1.5 Development of a Behavioral Fault Model

Examination of the relationship between the generalized functional fault model and the initial behavioral description will result in a behavioral fault model for the *if-then-else* construct in example IF1. This final step in the abstraction of *SSL* gate level faults into the behavioral domain provides the link between lower and higher level fault models, which has been lacking in previous research.

The *then* clause in example IF1, $Z \leq Y_0$, corresponds to the upper half of the truth table in Table 3-5. It can be seen from the fault table, that two of the functional faults, *SOP_A* and *POS_A*, uniquely affect the *then* clause. The *SOP* fault table in Table 3-3 shows that *SOP_A* was derived from the *SSL* gate level fault *G-1*. Referring to the gate level *SOP* circuit in Figure 3-3, it can be seen that the fault *G-1* causes undesired activation of *Channel B*, while attempting to select *Channel A*. The functional fault *SOP_A*, therefore, causes a corruption of *Channel A* by ORing it with *Channel B* in the final stage of the multiplexer. Behaviorally, this fault can be described as a corruption of the *then* clause, resulting in the definition of the behavioral fault *Clause-CORRUPT (OR)*, or specifically *THEN-CORRUPT (OR)*. The faulty version of the *then* clause can be modeled by ORing the right hand side of its assignment statement with the right hand side of the assignment statement from the *else* clause, resulting in $Z \leq Y_0 \text{ OR } Y_1$.

The other functional fault affecting the *then* clause, *POS_A*, was derived from the *SSL* gate level fault *F-0*, shown in Table 3-4. Referring to the gate level *POS* circuit in Figure 3-5, fault *F-0* also causes undesired activation of *Channel B*, while attempting to select *Channel A*. The functional fault *POS_A*, therefore, causes a corruption of *Channel A* by ANDing it with *Channel B* in the final stage of the multiplexer. Behaviorally, this fault also causes a corruption of the *then* clause, resulting in the definition of the behavioral fault *THEN-CORRUPT (AND)*. The faulty version of the *then* clause can be modeled by ANDing the right hand side of its assignment statement with the right hand side of the assignment statement from the *else* clause, resulting in $Z \leq Y_0 \text{ AND } Y_1$.

In a manner identical to the previous discussion, behavioral faults can be defined for the two functional faults that affect the *else* clause. The functional fault *SOP_B* corresponds to the behavioral fault *ELSE-CORRUPT (OR)*. Finally, the functional fault *POS_B* produces the behavioral fault *ELSE-CORRUPT (AND)*.

The generalized set of four functional faults derived for example IF1, therefore, result in the definition of four behavioral faults. These four faults form an initial behavioral fault model for the control construct *if-then-else*: Each clause of an *if-then-else* statement can be affected by two behavioral faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. The new control fault model is summarized in Figure 3-6.

THEN-CORRUPT (OR)	ELSE-CORRUPT (OR)
if SEL = '0' then	if SEL = '0' then
Z <= Y0 OR Y1;	Z <= Y0;
else	else
Z <= Y1;	Z <= Y1 OR Y0;
end if;	end if;
THEN-CORRUPT (AND)	ELSE-CORRUPT (AND)
if SEL = '0' then	if SEL = '0' then
Z <= Y0 AND Y1;	Z <= Y0;
else	else
Z <= Y1;	Z <= Y1 AND Y0;
end if;	end if;

Figure 3-6 Control fault model for if-then-else.

The test vectors for the new behavioral faults follow directly from the functional faults from which they were derived. The behavioral fault *THEN-CORRUPT (OR)* corresponds to the functional fault *SOP_A*. From Figure 3-6, the test vector (*SEL Y1 Y0*) 010, causes the *then* clause of the *if* statement to be selected. The fault free response assigns the 0 from input *Y0* to the output *Z*, regardless of the value on input *Y1*. Under the behavioral fault *THEN-CORRUPT (OR)*, the *then* clause assigns *Y0 OR Y1* to *Z*, resulting in *Z=1*, contrary to the fault free output. Thus, the test vector 010 constitutes a valid test for the behavioral fault *THEN-CORRUPT (OR)*.

The test vector for the behavioral fault *THEN-CORRUPT (AND)* comes from the functional fault *POS_A*, 001. The fault free behavior again selects the *then* clause, resulting in *Z=1*. Under the fault *THEN-CORRUPT (AND)*, the *then* clause assigns *Y0 AND Y1* to *Z*, resulting in *Z=0*. Similarly, the test vector for *ELSE-CORRUPT (OR)* is 101 and the test vector for *ELSE-CORRUPT (AND)* is 110.

3.1.6 Evaluation of the New Behavioral Fault Model

In order to evaluate the effectiveness of the new behavioral fault model, it will be compared to the underlying gate level faults it was meant to encompass. The four test vectors required to detect the behavioral faults in Figure 3-6 are 001, 010, 101, and 110. Applying these test vectors to the gate level circuits in Figure 3-3 (*SOP*) and Figure 3-5 (*POS*) results in the detection of the *SSL* gate level faults indicated in Table 3-6. Careful examination of the results confirms that the behavioral test vectors achieve complete coverage of *SSL* gate level faults in either the *SOP* or *POS* implementation of the multiplexer.

Behavioral fault	Test Vector	SOP SSL Faults Detected	POS SSL Faults Detected
THEN-CORRUPT (OR)	010	A-1, D-1, E-0, F-0, G-1, H-1, S-1, Z-1	A-1, C-1, D-1, S-1, Z-1
THEN-CORRUPT (AND)	001	A-0, C-0, D-0, E-0, S-1, Z-0	A-0, D-0, E-1, F-0, G-0, S-1, Z-0
ELSE-CORRUPT (OR)	101	B-1, C-1, D-1, E-1, H-1, S-0, Z-1	B-1, E-0, F-1, G-1, S-0, Z-1
ELSE-CORRUPT (AND)	110	B-0, E-1, F-1, G-0, H-0, S-0, Z-0	B-0, C-0, D-0, G-0, S-0, Z-0

Table 3-6 SSL faults detected by behavioral test vectors.

3.1.7 Comparison with Previous Behavioral Fault Models

As was discussed in Chapter 2 of this dissertation, most higher level fault models address the control construct *if-then-else*. The common fault model proposed by previous research is *stuck-then/stuck-else*. This fault model has been used by Armstrong [7][8][9] [13][58][69], Ghosh [18][27], Riesgo [60], and Chen [20][54]. All these researchers also use *stuck-data* faults on their non-control signals, so for completeness, these will be considered in combination with the control faults. Armstrong and Cho [22] also proposed the *Behavioral Stuck-at* (BSA) fault, combining *stuck-data* and *stuck-control* faults by defining *stuck-at* faults for unnamed signals corresponding to the conditional expressions of an *if* statement. All these behavioral fault models use the same fault technique, applying *stuck-at* faults to the data and control inputs of the circuit.

The effectiveness of the *stuck-data* and *stuck-control* fault models will now be evaluated by applying them to the behavioral description in example IF1. Possible test vector sets will be developed and their ability to detect SSL gate level faults will be compared. There are three data signals (Y_0 , Y_1 , Z) and one control signal (SEL) in the example. Applying the *stuck-data* and *stuck-control* fault models implies a *stuck-at-0* and *stuck-at-1* fault for each signal, resulting in the eight behavioral faults shown in Table 3-7. Note that the two control faults $SEL-0$ and $SEL-1$ are equivalent to *stuck-then* and *stuck-else*, respectively.

Test Vector	SEL	Y1	Y0	Z	Y0-0	Y0-1	Y1-0	Y1-1	Z-0	Z-1	SEL-0	SEL-1
Option 2	0	0	0	0		1				1		
Options 1 & 6	0	0	1	1	0				0		0	
Options 2 & 6	0	1	0	0		1				1	1	
Option 1	0	1	1	1	0				0			
Option 4	1	0	0	0				1		1		
Options 4 & 5	1	0	1	0				1		1	1	
Options 3 & 5	1	1	0	1			0		0		0	
Option 3	1	1	1	1			0		0			

Table 3-7 Stuck-data and stuck-control behavioral faults.

None of the test vectors in Table 3-7 are mandatory; each behavioral fault has at least two possible tests that detect it. For completeness, all possible combinations of test vectors will be examined. There are $2^6 = 64$ possible test vector sets, but due to overlapping coverage among test vectors, only 49 of them are unique.

First, consider the effectiveness of a test vector set of size four at detection of gate level *SSL* faults. The test vectors for Set 1 are listed in Table 3-8 along with the behavioral faults from Table 3-7 that they detect. As expected, all behavioral faults are detected. Also indicated in Table 3-8 are the *SOP* circuit gate level *SSL* faults, from Figure 3-3 and Table 3-1, which are detected by each of the test vectors. Examination of the gate level fault coverage shows that the *SOP SSL* faults *C-1*, *F-0*, and *G-1* are left undetected by test vector Set 1.

	Test Vector	Behavioral Faults Detected	SOP Gate Level SSL Faults Detected
Set 1	000	Y0-1, Z-1	A-1, D-1, H-1, Z-1
	001	Y0-0, Z-0, SEL-1	A-0, C-0, D-0, E-0, S-1, Z-0
	100	Y1-1, Z-1	B-1, D-1, H-1, Z-1
	110	Y1-0, Z-0, SEL-0	B-0, E-1, F-1, G-0, H-0, S-0, Z-0

Table 3-8 Faults detected by test vector Set 1.

Similar analysis shows that any test vector set that does not contain the test vector 101 will fail to detect the gate level fault *C-1*. Likewise, any test vector set that does not contain the test vector 010 will fail to detect the gate level faults *F-0* and *G-1*. Of the nine test vector sets of size four, only four sets contain both test vectors 010 and 101. Coverage does not improve greatly with test vector set size. Half of the 24 sets containing five test vectors do not contain both test vectors necessary to ensure complete gate level fault coverage. Even when the size of the test vector set is increased to six, there are still seven sets out of a possible 16 that fail to detect all gate level faults.

Similar results are obtained when the previous fault models are compared to a *POS* gate level circuit. A large number of possible test vector sets do not contain the test vectors necessary to ensure complete gate level fault coverage. Hence, the new behavioral fault model, based on functional analysis of control constructs, gives improved gate level fault coverage compared to the previous *stuck-then/stuck-else* fault model.

3.1.8 Expansion of the Fault Model

The previous example involved selection of one of two options, hence its implementation with a 2-to-1 multiplexer. Selection from a larger set of input options must now be examined and the effects on the behavioral fault model developed. The *if-then-else* construct contains an optional *elsif* clause to allow selection based on multiple conditions. Multiple *elsif* clauses can be used to allow selection among any number of inputs.

```

if SEL = "00" then
    Z <= Y0;
elsif SEL = "01" then
    Z <= Y1;
elsif SEL = "10" then
    Z <= Y2;
elsif SEL = "11" then
    Z <= Y3;
end if;

```

Figure 3-7 Behavioral description for example IF2.

Example IF2, in Figure 3-7, selects one of four inputs (Y_3, Y_2, Y_1, Y_0) for assignment to a single output (Z), based on the value of two control bits (SEL). For example, the control bits $SEL = "10"$ represent the binary encoding for 2, hence input Y_2 is selected for

assignment to Z. Note the lack of a final *else* clause, due to complete elaboration of values for SEL in the *if-elsif* clauses. Synthesis of example IF2 by AutoLogic II results in the expected 4-to-1 multiplexer architecture shown in Figure 3-8.

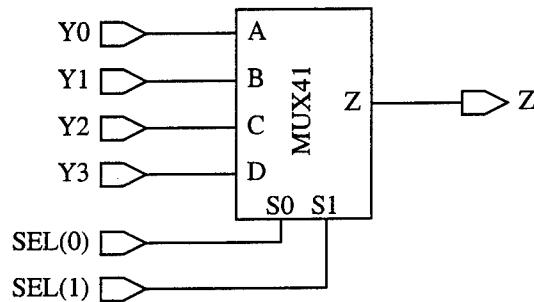


Figure 3-8 Synthesized hardware for example IF2.

3.1.8.1 Functional Analysis

In order to perform a functional analysis similar to example IF1, a gate level implementation of the entity MUX41 is needed. Mentor Graphics' Design Architect was used to provide the gate level detail shown in Figure 3-9. Note that the resulting SOP gate level structure agrees with the functional block diagram of the 4-line-to-1-line data selector/multiplexer found in *The TTL Data Book* [67].

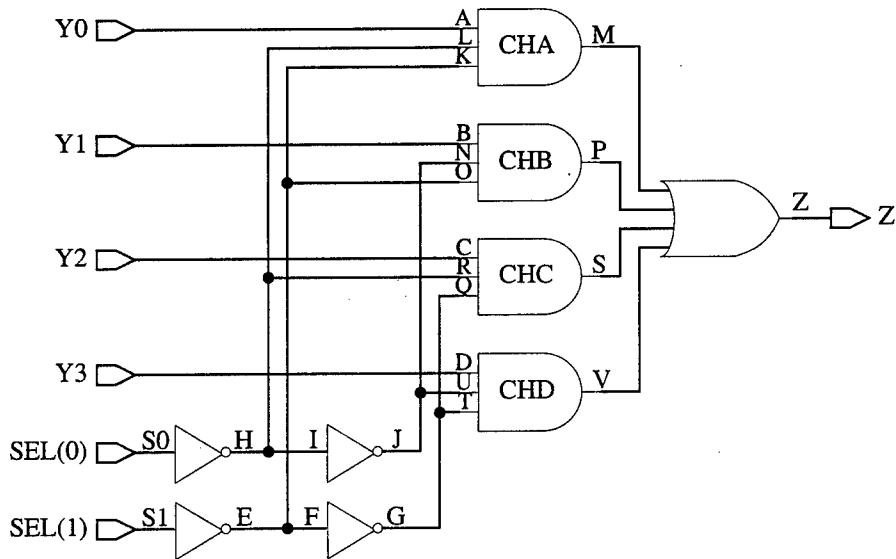


Figure 3-9 Gate level circuit for example IF2.

The gate level circuit contains 18 distinct internal lines (labeled E through V) in addition to the six inputs (A, B, C, D, S0, S1), and one output (Z) for a total of 25 logical lines.

Again, applying the SSL fault model where each line can be either *stuck-at-0* or *stuck-at-1*, there are a total of 50 gate level SSL faults in the circuit. The 2^6 possible input combinations result in a 64 by 50 fault table which will not be reproduced here. Reductions in the fault table due to *functional equivalence* and *fault dominance* are listed in Table 3-9.

Faults	Remarks	Faults	Remarks
A-1, I-0, J-1, P-1	Dominate N-1	K-0, L-0, M-0	Equivalent to A-0
B-1, M-1, Z-1	Dominate L-1	N-0, O-0, P-0	Equivalent to B-0
C-1, V-1	Dominate U-1	Q-0, R-0, S-0	Equivalent to C-0
D-1, S-1	Dominate R-1	T-0, U-0, V-0	Equivalent to D-0
I-1, J-0, Z-0	Dominate B-0		
F-0, G-1	Dominate Q-1		
F-1, G-0	Dominate C-0		

Table 3-9 Fault reductions for example IF2.

The gate level faults of primary interest are those controlling the switching of the multiplexer channels, the inputs to the channel AND gates. Activating the external control line fault $SEL(0)-0$ has the effect of simultaneously activating the faults $L-1$, $N-0$, $R-1$, and $U-0$. Each of these gate level faults corresponds to, or is equivalent to, an undominated functional fault. Testing for each of these functional faults provides complete test coverage for the fault $SEL(0)-0$. Combinations of undominated faults for each of the external control line faults are shown in Table 3-10.

Faults	Covering Faults			
S0-0, H-1	B-0	D-0	L-1	R-1
S0-1, H-0	A-0	C-0	N-1	U-1
S1-0, E-1	C-0	D-0	K-1	O-1
S1-1, E-0	A-0	B-0	Q-1	T-1

Table 3-10 Covering faults for external control line faults.

The resulting reduced fault set for example IF2 contains three distinct faults for each of the four channels of the multiplexer, for a total of 12 functional faults. The set of func-

tional faults is summarized in Table 3-11 along with the associated test vectors; don't care values are indicated by an X. For ease of comparison with the remarks column, the vectors are labeled with the multiplexer inputs ($S_1 S_0 D C B A$). This notation for test vectors is equivalent to the external input combination ($SEL Y_3 Y_2 Y_1 Y_0$).

Fault	Remarks	Test Vector ($S_1 S_0 D C B A$)
A-0	$CHA = 0$	00 XXX1
B-0	$CHB = 0$	01 XX1X
C-0	$CHC = 0$	10 X1XX
D-0	$CHD = 0$	11 1XXX
N-1	$CHA = CHA \text{ OR } CHB$	00 XX10
Q-1	$CHA = CHA \text{ OR } CHC$	00 X1X0
L-1	$CHB = CHB \text{ OR } CHA$	01 XX01
T-1	$CHB = CHB \text{ OR } CHD$	01 1X0X
K-1	$CHC = CHC \text{ OR } CHA$	10 X0X1
U-1	$CHC = CHC \text{ OR } CHD$	10 10XX
O-1	$CHD = CHD \text{ OR } CHB$	11 0X1X
R-1	$CHD = CHD \text{ OR } CHC$	11 01XX

Table 3-11 Reduced functional faults for example IF2.

Examination of the remarks column of Table 3-11, indicates that each channel can be corrupted by two different sources. Each of a channel's logically adjacent neighbors can cause a corruption, where logical adjacency means that the combination of control inputs ($S_1 S_0$) differ by only one bit. For example, *Channel A* is selected by control inputs 00 and is, therefore, logically adjacent to *Channel B* (01) and *Channel C* (10).

Further examination of the test vectors associated with any pair of corruptions shows that, due to the don't cares, these vectors are not inconsistent with each other. It is possible to chose a single test vector which will detect both corruptions of a channel by its adjacent neighbors. Using the terminology of Hansen and Hayes [29], a *compatible fault set* is defined as a set of functional faults that can be detected by a single test. Hence, the two corruptions of a channel by its logically adjacent neighbors form a *compatible fault set*.

3.1.8.2 Generalized Functional Fault Model

Comparison of the reduced functional faults for the *SOP* 4-to-1 multiplexer with a corresponding set of functional faults for a *POS* implementation yields a generalized functional fault model, not tied to a specific realization. Recall from Sections 3.1.4 and 3.1.5 that the channel corruptions originating in an *SOP* realization produced an ORing of adjacent channels, while a *POS* circuit caused an ANDing of those channels. A generalized set of functional faults and their corresponding test vectors are presented in Table 3-12. Note the combination of test vectors to form *compatible fault sets*.

Functional Fault	Remarks	Test Vector (S1 S0 D C B A)
CHA-CORRUPT (by CHB)(AND)	CHA = CHA AND CHB	00 X001
CHA-CORRUPT (by CHC)(AND)	CHA = CHA AND CHC	
CHB-CORRUPT (by CHA)(AND)	CHB = CHB AND CHA	01 0X10
CHB-CORRUPT (by CHD)(AND)	CHB = CHB AND CHD	
CHC-CORRUPT (by CHA)(AND)	CHC = CHC AND CHA	10 01X0
CHC-CORRUPT (by CHD)(AND)	CHC = CHC AND CHD	
CHD-CORRUPT (by CHB)(AND)	CHD = CHD AND CHB	11 100X
CHD-CORRUPT (by CHC)(AND)	CHD = CHD AND CHC	
CHA-CORRUPT (by CHB)(OR)	CHA = CHA OR CHB	00 X110
CHA-CORRUPT (by CHC)(OR)	CHA = CHA OR CHC	
CHB-CORRUPT (by CHA)(OR)	CHB = CHB OR CHA	01 1X01
CHB-CORRUPT (by CHD)(OR)	CHB = CHB OR CHD	
CHC-CORRUPT (by CHA)(OR)	CHC = CHC OR CHA	10 10X1
CHC-CORRUPT (by CHD)(OR)	CHC = CHC OR CHD	
CHD-CORRUPT (by CHB)(OR)	CHD = CHD OR CHB	11 011X
CHD-CORRUPT (by CHC)(OR)	CHD = CHD OR CHC	

Table 3-12 Generalized functional faults for example IF2.

3.1.8.3 Behavioral Fault Model

Examination of the relationship between the generalized set of functional faults and the initial behavioral description for example IF2 will result in a behavioral fault model for the expanded *if-then-else* construct.

The first *then* clause, $Z \leq Y_0$, is activated when $SEL = "00"$, corresponding to *Channel A* of the multiplexer. This clause will be referred to as the *00-THEN* clause. From the functional analysis of the multiplexer in Section 3.1.8.2, *Channel A* was affected by four functional faults, *CHA-CORRUPT* (by *CHB*)(AND), *CHA-CORRUPT* (by *CHC*)(AND), *CHA-CORRUPT* (by *CHB*)(OR), and *CHA-CORRUPT* (by *CHC*)(OR). These faults can be mapped into the behavioral domain just like those from example IF1.

The functional fault *CHA-CORRUPT* (by *CHB*)(AND) represents corruption of *Channel A* by *Channel B*, which maps directly to the behavioral fault *00-THEN-CORRUPT* (by *01*)(AND). The faulty version of the *then* clause can be represented by ANDing the right hand side of the assignment statement corresponding to *Channel A* with the right hand side of the assignment statement from *Channel B*. The resulting faulty version of the assignment statement for the *00-THEN* clause becomes $Z \leq Y_0 \text{ AND } Y_1$. Likewise, the corruption of *Channel A* by *Channel C* results in the definition of the behavioral fault *00-THEN-CORRUPT* (by *10*)(AND).

The (*OR*) corruptions of the *00-THEN* clause are formed in a similar manner. Finally, the remaining *then* clauses each have four behavioral faults, corresponding to their respective channels. A subset of the resulting 16 behavioral faults, four for each of the four clauses/channels, is presented in Figure 3-10.

```
00-THEN-CORRUPT (by 01)(OR)
  if SEL = "00" then
    Z <= Y0 OR Y1;
  elseif SEL = "01" then
    Z <= Y1;
  elseif SEL = "10" then
    Z <= Y2;
  elseif SEL = "11" then
    Z <= Y3;
  end if;
```

```
00-THEN-CORRUPT (by 10)(OR)
  if SEL = "00" then
    Z <= Y0 OR Y2;
  elseif SEL = "01" then
    Z <= Y1;
  elseif SEL = "10" then
    Z <= Y2;
  elseif SEL = "11" then
    Z <= Y3;
  end if;
```

Figure 3-10 Control fault model for expanded if-then-else.

<i>01-THEN-CORRUPT (by 00)(OR)</i>	<i>01-THEN-CORRUPT (by 11)(OR)</i>
<pre> if SEL = "00" then Z <= Y0; elsif SEL = "01" then Z <= Y1 OR Y0; elsif SEL = "10" then Z <= Y2; elsif SEL = "11" then Z <= Y3; end if; </pre>	<pre> if SEL = "00" then Z <= Y0; elsif SEL = "01" then Z <= Y1 OR Y3; elsif SEL = "10" then Z <= Y2; elsif SEL = "11" then Z <= Y3; end if; </pre>

Figure 3-10 Control fault model for expanded if-then-else.

3.1.8.4 Evaluation of the Behavioral Fault Model

To evaluate the effectiveness of the behavioral fault model for the expanded *if-then-else* construct, it will be compared to the underlying *SSL* gate level faults it was meant to encompass. Test vectors will be determined for each behavioral fault and the gate level fault coverage of these test vectors will be examined.

Testing for the first behavioral fault from Figure 3-10, *00-THEN-CORRUPT (by 01)(OR)*, requires activation of the first *then* clause with $SEL = "00"$. The fault free behavior, $Z \leq Y0$, must be compared to the faulty behavior, $Z \leq Y0 \text{ OR } Y1$. Setting $Y0=0$ will result in $Z=0$ for an uncorrupted channel. If $Y1=1$, then a corruption of the channel by the adjacent channel, *01-THEN*, will result in $Z=1$, thus detecting the fault. Since the other two clauses do not cause any corruptions with this fault, the other inputs ($Y3, Y2$) can remain don't cares. The resulting test vector ($SEL\ Y3\ Y2\ Y1\ Y0$) is $00XX10$. By similar analysis, the other (*OR*) corruption fault for this clause, *00-THEN-CORRUPT (by 10)(OR)*, requires the test vector $00X1X0$.

Recall that the functional faults for *Channel-CORRUPT* formed a compatible fault set, one that could be detected by a single test vector. Likewise, the two *Clause-CORRUPT* behavioral faults should also form a compatible fault set. The previous analysis determined that the two test vectors necessary to detect the two *Clause-CORRUPT (OR)* faults for the *00-THEN* clause were $00XX10$ and $00X1X0$. Comparison of the two test vectors confirms that they are, in fact, compatible. Elimination of don't cares for $Y2$ and $Y1$ produces a combined test vector $00X110$. The behavioral faults and their corresponding test vectors are summarized in Table 3-13.

Behavioral Fault	Test Vector (SEL Y3 Y2 Y1 Y0)
00-THEN-CORRUPT (by 01)(AND)	00 X001
00-THEN-CORRUPT (by 10)(AND)	
01-THEN-CORRUPT (by 00)(AND)	01 0X10
01-THEN-CORRUPT (by 11)(AND)	
10-THEN-CORRUPT (by 00)(AND)	10 01X0
10-THEN-CORRUPT (by 11)(AND)	
11-THEN-CORRUPT (by 01)(AND)	11 100X
11-THEN-CORRUPT (by 10)(AND)	
00-THEN-CORRUPT (by 01)(OR)	00 X110
00-THEN-CORRUPT (by 10)(OR)	
01-THEN-CORRUPT (by 00)(OR)	01 1X01
01-THEN-CORRUPT (by 11)(OR)	
10-THEN-CORRUPT (by 00)(OR)	10 10X1
10-THEN-CORRUPT (by 11)(OR)	
11-THEN-CORRUPT (by 01)(OR)	11 011X
11-THEN-CORRUPT (by 10)(OR)	

Table 3-13 Behavioral test vectors for example IF2.

The behavioral test vectors from Table 3-13 provide complete *SSL* gate level fault coverage for both *SOP* and *POS* implementations of example IF2. The results confirm that the control fault model for the expanded *if-then-else* construct is a valid abstraction into the behavioral domain of the original gate level *SSL* faults.

3.1.9 Summary

A new behavioral fault model has been developed for the control construct *if-then-else*. This new control fault model specifies that each clause of an *if-then-else* statement can be affected by two different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. The actual number of *Clause-CORRUPT* faults depends on the size of the model and the resulting number of logical adjacencies between clauses.

In the VHDL behavioral description, a *Clause-CORRUPT (OR)* fault is represented by considering the effect of the corrupting clause. The right hand side of the assignment statement for the corrupted clause is ORed with the right hand side of the assignment statement for the corrupting clause. A test vector for this fault is determined by setting the fault free behavior of the uncorrupted clause to '0'. The corrupting clause is then set to '1', thus producing a conflict with the fault free behavior. Multiple *Clause-CORRUPT (OR)* faults may affect the same clause, due to logical adjacencies between clauses. These faults can form a compatible fault set and their test vectors can, therefore, be combined to produce a single test for the corruption of that clause.

The faulty operation of a clause due to a *Clause-CORRUPT (AND)* fault is represented in a similar manner. The right hand side of the assignment statement for the corrupted clause is ANDed with the right hand side of the assignment statement for the corrupting clause. A test vector forces the fault free behavior of the uncorrupted clause to produce an output of '1', while the corrupting clause is set to produce an output of '0'.

Test vectors generated from these behavioral faults can be applied to gate level implementations of the behavioral descriptions. Examples have shown the ability of these test vectors to detect underlying gate level *SSL* faults in synthesized circuits. Analysis has also shown that the test vectors from this new control fault model do provide improved gate level fault coverage over previous behavioral fault models.

3.2 CASE

The other VHDL control construct closely related to *if-then-else* is the *case* statement. The *case* statement allows selection of statements to execute depending on the value of a selection *expression*. Multiple *when* clauses can be used to allow selection among any number of *choices*. All *choices* must be distinct and all values must be represented in the *choice* lists, or the special *choice others* must be included as the last alternative.

The *case* statement is really just an alternative representation of the *if-then-else* construct with more restrictions. Use of *if-then-else* versus *case* is usually just a matter of programming style; any *case* statement can be rewritten as an equivalent *if-then-else*. The example in Figure 3-11 shows two equivalent behavioral descriptions, one using an *if* statement and the other using a *case* statement. Note that the *when* clauses perform the same function for the *case* statement as the *then* clauses in the *if* statement. The *when oth-*

ers clause ensures complete elaboration of *choices* for the *case* statement, just like the *else* clause for the *if* statement.

```

        case SEL is
    if SEL = "00" then      when "00" =>
        Z <= Y0;           Z <= Y0;
    elsif SEL = "01" then   when "01" =>
        Z <= Y1;           Z <= Y1;
    else                   when others =>
        Z <= Y2;           Z <= Y2;
    end if;                end case;

```

Figure 3-11 Equivalent if-then-else and case statements.

In Appendix E Armstrong and Gray and Mentor Graphics demonstrate the link between the *case* statement and the functional building block of a multiplexer. These examples reinforce the similarities between the control constructs *if-then-else* and *case*.

3.2.1 Application of the Control Fault Model

The simple example, CASE1, from Figure 3-11 will be used to demonstrate the application of the new control fault model to the *case* construct. The fault model specifies that each clause of an *if-then-else* statement, and hence of a *case* statement, can be affected by two different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. The actual number of *Clause-CORRUPT* faults depends on the size of the model and the resulting number of logical adjacencies between clauses. *Clause-CORRUPT* faults are now designated *WHEN-CORRUPT* faults, identified by the appropriate *choice* as well as the corrupting clause.

In order to specify the *WHEN-CORRUPT* faults, the logical adjacencies between clauses must be identified. Figure 3-12 provides a graphical representation of the logical adjacencies between the clauses for example CASE1. The *when others* clause defines the behavior for all *choices* not explicitly specified in previous clauses. Figure 3-12 indicates that each clause of the *case* statement is adjacent to two other clauses. Hence, there are two *WHEN-CORRUPT (OR)* faults and two *WHEN-CORRUPT (AND)* faults for each of the three clauses in the example.

		SEL(1)	
		0	1
SEL(0)	0	Y0	Y2
	1	Y1	
		Z	

Figure 3-12 Logical adjacencies among clauses.

The *when others* clause from example CASE1 corresponds to the control input combination $SEL = "1X"$. In order to specify an adjacency between the *when others* clause and another clause, the don't care must be eliminated. For example, the clause *when "00"* should be logically adjacent to the clauses *when "01"* and *when "10"*. Though the clause *when "10"* does not explicitly exist, it is created by the designation of the don't care for the *when others* clause as $SEL(0)=0$. The resulting adjacencies and *WHEN-CORRUPT (AND)* faults are shown in Figure 3-13. An additional six behavioral faults for *WHEN-CORRUPT (OR)* faults are determined in a similar manner.

WHEN-00-CORRUPT (by 01)(AND)

```
case SEL is
  when "00" =>
    Z <= Y0 AND Y1;
  when "01" =>
    Z <= Y1;
  when others =>
    Z <= Y2;
end case;
```

WHEN-01-CORRUPT (by 00)(AND)

```
case SEL is
  when "00" =>
    Z <= Y0;
  when "01" =>
    Z <= Y1 AND Y0;
  when others =>
    Z <= Y2;
end case;
```

WHEN-00-CORRUPT (by 10)(AND)

```
case SEL is
  when "00" =>
    Z <= Y0 AND Y2;
  when "01" =>
    Z <= Y1;
  when others =>
    Z <= Y2;
end case;
```

WHEN-01-CORRUPT (by 11)(AND)

```
case SEL is
  when "00" =>
    Z <= Y0;
  when "01" =>
    Z <= Y1 AND Y2;
  when others =>
    Z <= Y2;
end case;
```

Figure 3-13 WHEN-CORRUPT (AND) faults for example CASE1.

WHEN-10-CORRUPT (by 00)(AND) <pre> case SEL is when "00" => Z <= Y0; when "01" => Z <= Y1; when others => Z <= Y2 AND Y0; end case;</pre>	WHEN-11-CORRUPT (by 01)(AND) <pre> case SEL is when "00" => Z <= Y0; when "01" => Z <= Y1; when others => Z <= Y2 AND Y1; end case;</pre>
---	---

Figure 3-13 WHEN-CORRUPT (AND) faults for example CASE1.

Test vectors for the behavioral faults are derived using the methodology described in Section 3.1.9. Recall that vectors derived from corruptions to the same channel can form a *compatible fault set* with a single test. The resulting test vectors are listed in Table 3-14.

Behavioral Fault	Test Vector (SEL Y2 Y1 Y0)
WHEN-00-CORRUPT (by 01)(AND)	00 001
WHEN-00-CORRUPT (by 10)(AND)	
WHEN-01-CORRUPT (by 00)(AND)	01 010
WHEN-01-CORRUPT (by 11)(AND)	
WHEN-10-CORRUPT (by 00)(AND)	10 1X0
WHEN-11-CORRUPT (by 01)(AND)	11 10X
WHEN-00-CORRUPT (by 01)(OR)	00 110
WHEN-00-CORRUPT (by 10)(OR)	
WHEN-01-CORRUPT (by 00)(OR)	01 101
WHEN-01-CORRUPT (by 11)(OR)	
WHEN-10-CORRUPT (by 00)(OR)	10 0X1
WHEN-11-CORRUPT (by 01)(OR)	11 01X

Table 3-14 Behavioral test vectors for example CASE1.

3.2.2 Evaluation of the Fault Model

The VHDL behavioral description for example CASE1 was synthesized using Mentor Graphics' AutoLogic II. The resulting multiplexer architecture is shown in Figure 3-14.

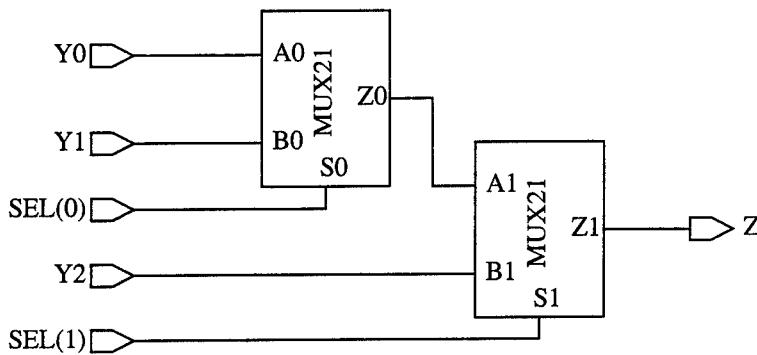


Figure 3-14 Synthesized Structure1 for example CASE1.

Fault simulations were then performed on the gate level circuit for Structure1 using the behavioral test vectors shown in WAVES format in Figure 3-15.

```
% SEL Y2 Y1 Y0 Z : time ;
% Clause-CORRUPT(AND)
00 001 1 : 500 ns ;
01 010 1 : 500 ns ;
10 1X0 1 : 500 ns ;
11 10X 1 : 500 ns ;
% Clause-CORRUPT(OR)
00 110 0 : 500 ns ;
01 101 0 : 500 ns ;
10 0X1 0 : 500 ns ;
11 01X 0 : 500 ns ;
```

Figure 3-15 WAVES test vectors for example CASE1.

According to MIL-STD 883D [52], Structure1 contains 34 unique gate level SSL faults. Simulations are performed for each of the gate level faults and the fault is reported as detected when the circuit's output differs from the expected output. All SSL faults were detected by the behavioral test vectors in the multiplexer implementation of example CASE1. The fault coverage was next evaluated for an alternate gate level realization.

AutoLogic II was again used to synthesize and then optimize the VHDL behavioral description for example CASE1. The resulting circuit for Structure2 is shown in Figure 3-16. Note that the circuit contains a mix of NAND, OR, and NOT gates and does not directly relate to any of the circuits analyzed in the development of the behavioral fault models for the control constructs *if-then-else* or *case*.

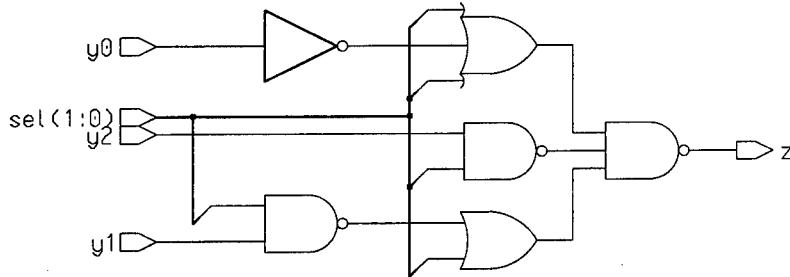


Figure 3-16 Synthesized Structure2 for example CASE1.

Fault simulations were performed on the gate level circuit using the behavioral test vectors from Figure 3-15. The resulting fault coverage as a function of the input test vectors is shown in Figure 3-17. Though the actual shape of the graph may vary with the order of the application of the test vectors, the resulting fault coverage will be the same. Fault coverage for Structure2 was $34/34 = 100\%$, complete SSL gate level fault coverage.

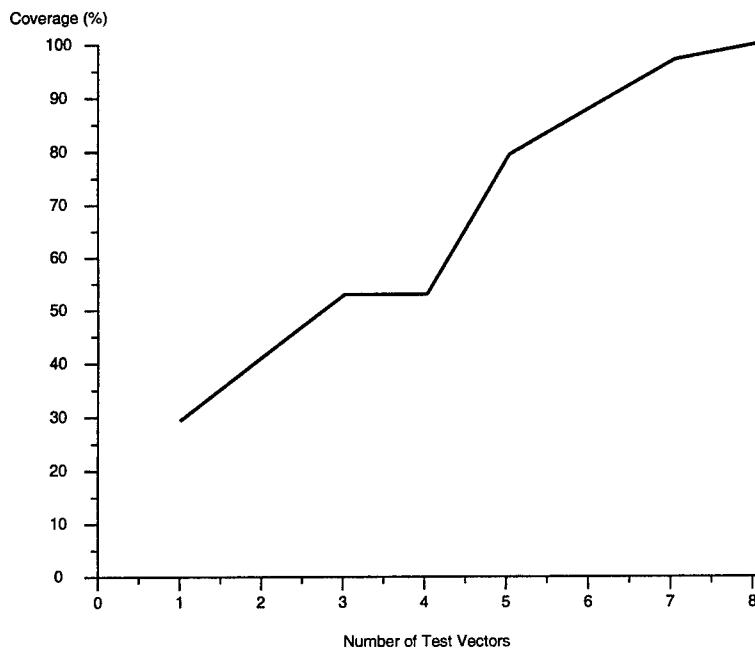


Figure 3-17 Fault coverage for Structure2 of example CASE1.

3.2.3 Comparison with Previous Behavioral Fault Models

As was discussed in Chapter 2 of this dissertation, most higher level fault models also address the control construct *case*. However, contrary to *if-then-else* where there was a general consensus, varying fault models are proposed for the *case* statement. Armstrong [7][8][9][13][58][69] proposes the *dead clause* fault in which each clause in a *case* state-

ment fails to a no-operation. The *dead clause* fault model is also used by Chen [20][54]. Armstrong and Cho [22] later define the Behavioral Stuck-open (*BSO*) fault, combining *assignment control* faults and *dead clause* faults. In a *BSO* fault the value of the source expression of an assignment statement is not correctly transferred to its target.

Since Ghosh [18][27] bases his fault models on the programming language C, instead of VHDL, the equivalent to the *case* statement is the *switch (Id)*. The corresponding fault model states that the *switch* may select either the cases corresponding to the lower or upper extremes of the *switch* identifier's value system, all, or none of the specified cases. Finally, Riesgo [60] proposes a fault model for *case-is-when* in which the expression which controls the *case* statement may be *stuck-at-“all possible values.”*

Recall that the new behavioral fault model for the *case* statement was based on the fact that *if-then-else* and *case* represent similar selection activities leading to multiplexer architectures. Any *case* statement can be rewritten as an equivalent *if-then-else*, therefore, the faults models for the equivalent statements should be the same. Only Riesgo proposes a fault model for the *case* statement which is equivalent to the one proposed for *if-then-else*. The equivalent *stuck-control* fault model is *stuck-then/stuck-else*, which was shown to have deficiencies in detecting gate level faults by the analysis in Section 3.1.7.

Ghosh's fault model is based on multiple *stuck-at* faults on the control inputs resulting in only the upper and lower extremes of the *switch (Id)* statement. Individual *stuck-at* faults are not considered on control lines, causing intermediate values to be neglected. Hence, faults developed for a *switch* statement, will not be consistent with those developed for an equivalent *if-then-else*. Also, the hardware analogy, that the decoder for *Id* may fail such that a logic 1 is always asserted at all of the output ports, would correspond to the unlikely scenario of all the *channels* of the multiplexer being simultaneously *stuck-on*.

The *dead clause* and *BSO* faults proposed by Armstrong represent the farthest divergence from those proposed for *if-then-else*. This occurs even though Armstrong and Gray [10] demonstrate the similarities between *if-then-else* and *case* in their discussion of hardware implementation of VHDL constructs. The faulty version of a clause is represented by replacing the right hand side (source) of an assignment statement with the left hand side (target). For example, an assignment statement from example CASE2, $Z \leftarrow Y_1$,

would be faulted as $Z \leq Z$. The result is inconsistent with the faults that would be defined for an equivalent *if-then-else* construct.

3.3 Conclusions

A new behavioral fault model has been developed for the control constructs *if-then-else* and *case*. The new fault model is more closely linked to underlying hardware by combining VHDL synthesis information with the industry standard *single-stuck-line* (*SSL*) fault model. Each clause of an *if-then-else* or *case* statement can be affected by 2 different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. The actual number of *Clause-CORRUPT* faults depends on the size of the model and the resulting number of logical adjacencies between clauses.

Test vectors derived from these control faults can be applied to gate level implementations of the VHDL behavioral descriptions. Examples have shown the ability of these test vectors to detect underlying gate level *SSL* faults in synthesized circuits. Because the behavioral faults were derived from a functional analysis of the selection activity of multiplexers, they provide complete gate level fault coverage over a broad range of implementations. Detailed analysis has shown that the test vectors from this new control fault model do provide improved gate level fault coverage over previous behavioral fault models.

The new control fault model developed in this chapter provides two improvements over previous behavioral fault models. First, detection of low level *SSL* faults is improved by linking the selection activity of control constructs to the functional building block of a multiplexer. Finally, the control constructs *if-then-else* and *case* have been brought together in a single consistent fault model, where each clause can be affected by a some number of *Clause-CORRUPT* faults.

Chapter 4

Relational Operators

The only *relational operator* used thus far has been the “=” as part of the *condition* which controlled the *if* statement. The inclusion of other *relational operators* such as “>” and “<” implies the use of a comparator module in hardware. The comparison function will first be analyzed, using the techniques developed for the multiplexing function, in order to determine a generalized set of functional faults. These faults will then be abstracted into the behavioral domain by examination of the relationship between the functional faults and the behavior of the *relational operators*.

Because of similarities in their functions, the predefined VHDL *relational operators* from Table 1-2 can be divided into two groups for analysis. The relation $A > B$ (*GT*) is the same as $B < A$ (*LT*), hence only one of these operators needs to be analyzed. The relation $A > B$ (*GT*) is the opposite of $A \leq B$ (*LE*); the hardware need only differ by a single inverter. Likewise $A < B$ (*LT*) is related to $A \geq B$ (*GE*). Therefore, the set of generalized functional faults developed for the *GT* function can be used as a basis for all of the above relations (*GT, LT, GE, LE*). The same argument implies that the relation $A = B$ (*EQ*) can be analyzed to find functional faults which also apply to the function $A \neq B$ (*NE*).

4.1 Greater Than (GT)

Several common implementations of the *GT* function were first analyzed to assess the “worst case” for functional faults. Abstraction of these faults into the behavioral domain should provide complete gate level *SSL* fault coverage over a broad range of possible realizations of the comparison functions. Gate level circuits were examined for 2-level (*SOP* and *POS*), factored, and cascade implementations.

A comparison of functional faults indicates that the test vectors required to cover all gate level faults in either the factored or cascade implementations are a subset of the vectors for the 2-level realizations. Furthermore, a common set of test vectors provides complete gate level fault coverage in either the *SOP* or *POS* circuits. Therefore, just like the analysis of the multiplexing function, the *SOP* and *POS* implementations will again be used to form a generalized set of functional faults for the *GT* function.

4.1.1 Generalized Functional Faults

The *SOP* and *POS* implementations of the 2-bit $A > B$ function can be obtained from the analysis of the Karnaugh map shown in Figure 4-1.

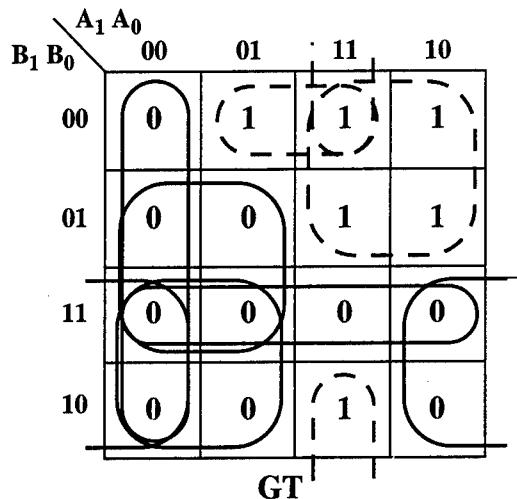


Figure 4-1 Karnaugh map for 2-bit GT function.

The groupings of minterms are shown with dashed lines and produce the *SOP* function $GT = A_1 \bar{B}_1 + A_0 \bar{B}_1 \bar{B}_0 + A_1 A_0 \bar{B}_0$. A functional analysis of the gate level *SOP* circuit produces a reduced set of functional faults shown in Table 4-1. Test vectors are shown in base-4 for ease of magnitude comparison of A and B .

Fault	Faulty Output	Test Vectors (AB)
G-1	1	00
G-0	0	10
I-1	1	11
H-1	1	12
E-0	0	20, 21, 31
K-1	1	22
J-0	0	32
L-1	1	33

Table 4-1 Reduced functional faults for SOP GT.

Note that a faulty output of 0 occurs when A is in fact greater than B (e.g. 32), yet the gate level circuit fails to give a proper indication of $GT = 1$. Likewise a faulty output of 1 occurs when A is not greater than B (e.g. 22).

Grouping of the maxterms in the Karnaugh map in Figure 4-1, shown with solid lines, produces the *POS* function $GT = (A_1 + A_0)(A_1 + \overline{B}_1)(A_1 + \overline{B}_0)(A_0 + \overline{B}_1)(\overline{B}_1 + \overline{B}_0)$. A functional analysis of the gate level *POS* circuit produces the reduced set of functional faults shown in Table 4-2.

Fault	Faulty Output	Test Vectors
E-1	1	00
F-0	0	10
I-1	1	11
G-1	1	12
E-0	0	20, 21
I-0	0	21, 31
K-1	1	22
G-0	0	32
M-1	1	33

Table 4-2 Reduced functional faults for POS GT.

Examination of the two sets of functional faults indicates that the seven mandatory test vectors are the same for both the *SOP* and *POS* circuits. Note that a single input combination (21) will detect the remaining faults in either implementation. Thus, a set of eight generalized functional faults and test vectors is produced.

4.1.2 Classification of Functional Faults

In order to aid in the analysis and classification of the generalized set of functional faults, an alternate representation is shown in Figure 4-2. The columns of the figure indicate the base-4 value of input A , while the rows specify the value of B . The heavy line cutting through the figure represents the GT function; the shaded area above the line indicates

A is greater than B . The diagonal just below the heavy line indicates where A and B are equal.

		A			
		0	1	2	3
B		0	I	III	
		1		I	III
		2		II	I
		3			I

Figure 4-2 Fault classes for 2-bit GT function.

The roman numerals in Figure 4-2 indicate the location of the test vectors for the eight functional faults derived in Section 4.1.1. Class I has been assigned to the test vectors where A and B are equal ($00, 11, 22, 33$). For this class, the faulty function gives an erroneous *TRUE* for the relation $A > B$. The other fault that produces an erroneous *TRUE* (12) has been designated as Class II. In this class, A is less than, rather than equal to, B . Finally, Class III has been assigned to the test vectors where A is in fact greater than B ($10, 21, 32$), but the faulty function gives an erroneous *FALSE*.

Classes I and III follow easily identifiable patterns, however, Class II is still somewhat vague. To gain additional insight into the make-up of Class II and to ensure that the initial classification of functional faults is valid, a functional analysis was performed on a 3-bit *GT* function. The resulting functional faults are classified in Figure 4-3.

As expected, the functional faults for Classes I and III follow the same patterns as the 2-bit case. The octal test vectors in Group II ($12, 34, 56$) provide the additional information necessary to identify a pattern. Note that in both Classes II and III, A and B differ by only 1; $A + 1 = B$ or $A = B + 1$ respectively (+ indicates addition). By induction, these fault classifications can now be generalized for an n -bit *GT* function. The resulting functional faults and fault classes are presented in Table 4-3.

		A							
		0	1	2	3	4	5	6	7
B	0	I	III						
	1		I	III					
	2		II	I	III				
	3				I	III			
	4				II	I	III		
	5						I	III	
	6						II	I	III
	7								I

Figure 4-3 Fault classes for 3-bit GT function.

Class	Faulty Output	A vs. B	# Test Vectors
I	TRUE	$A = B$	2^n
II	TRUE	$A + 1 = B$ (A odd)	$2^{n-1} - 1$
III	FALSE	$A = B + 1$	$2^n - 1$

Table 4-3 Functional faults for n-bit GT.

4.1.3 Behavioral Fault Model

Now that the three classes of functional faults have been identified, they can be abstracted into the behavioral domain by examining the relationship between the faults and the original VHDL operator ($>$). Appendix C gives the details of usage for *relational operators* in *expressions*, which form the *conditions* for the *if* statement. The *expressions* yield Boolean results, which control the selection of the appropriate clause. In order to model a fault in a *relational operator*, the controlling *expression* needs to be modified such that it produces an erroneous result (*TRUE* or *FALSE*) corresponding to that fault.

First consider the four Class I faults for a 2-bit *GT* function. The fault-free controlling *condition* can be written as:

```
if A > B then
```

Class I faults produce an erroneous *TRUE* when $A = B$, so the initial inclination would be to simply modify the *condition* to read:

```
if (A > B) OR (A = B) then
```

However, this has the effect of producing a single behavioral fault which can be detected by any one of the four required test vectors. What is needed is a distinct behavioral fault for each test vector. Hence, the first Class I behavioral fault can be rewritten as:

```
if (A > B) OR ((A = B) AND (A = "00")) then
```

The other three Class I faults are modeled by enumeration of the appropriate values for *A*.

Class II faults also produce an erroneous *TRUE*, this time when $A + 1 = B$ (*A* odd). Hence the single Class II fault for the 2-bit GT function can be modeled as:

```
if (A > B) OR ((A + 1 = B) AND (A = "01")) then
```

Finally, Class III faults produce an erroneous *FALSE* when $A = B + 1$. To model these faults in the behavioral domain, the original expression can be ANDed with 0 for the appropriate input combination. The NAND function produces the required behavior, giving the following model for the first Class III fault:

```
if (A > B) AND ((A = B + 1) NAND (B = "00")) then
```

The complete set of Class III faults is modeled by enumeration of the $2^n - 1$ values for *B*.

Class	Faulty Expression	Faults
I	(A > B) OR ((A = B) AND (A = "00"))	A = "00" A = "01" A = "10" A = "11"
II	(A > B) OR ((A + 1 = B) AND (A = "01"))	A = "01"
III	(A > B) AND ((A = B + 1) NAND (B = "00"))	B = "00" B = "01" B = "10"

Table 4-4 Behavioral faults for 2-bit GT.

4.1.4 Adapting the Model for GE, LT, and LE

The three fault classes and their associated behavioral fault models can be easily adapted for the other *relational operators* in this group (*GE*, *LT*, *LE*). As an example, the functional faults for the relation $A \geq B$ (*GE*) are presented in Figure 4-4. The thick line representing the *GE* function now lies below the diagonal of the figure. The locations of the Class I faults remain unchanged, however they now represent an erroneous *FALSE* for the *GE* function. Class II faults still produce the same erroneous result as Class I faults and the Class III faults now result in an incorrect *TRUE*.

		A			
		0	1	2	3
B	0	I			
	1	III	I	II	
	2		III	I	
	3			III	I

Figure 4-4 Fault classes for 2-bit GE function.

The functional faults for the *GE* function can now be summarized in Table 4-5 and result in the definition of the behavioral faults in Table 4-6.

Class	Faulty Output	A vs. B	# Test Vectors
I	FALSE	$A = B$	2^n
II	FALSE	$A = B + 1$ (B odd)	$2^{n-1} - 1$
III	TRUE	$A + 1 = B$	$2^n - 1$

Table 4-5 Functional faults for n-bit GE.

Similar analysis produces the generalized functional faults for the *LE* and *LT* functions. Since these functions differ by only a single inverter from *GT* and *GE* respectively, the behavioral fault models follow directly by inversion of the faulty output values.

Class	Faulty Expression	Faults
I	(A >= B) AND ((A = B) NAND (A = "00"))	A = "00" A = "01" A = "10" A = "11"
II	(A >= B) AND ((A = B + 1) NAND (B = "01"))	B = "01"
III	(A >= B) OR ((A + 1 = B) AND (A = "00"))	A = "00" A = "01" A = "10"

Table 4-6 Behavioral faults for 2-bit GE.

4.1.5 Summary

A set of functional and behavioral faults have been developed for the first group of relational operators (*GT*, *GE*, *LT*, *LE*). Each operator is affected by three classes of faults. Class I faults occur when the *operands* of the relation are equal. Class II faults produce the same faulty output as Class I faults. Finally, Class III faults produce the opposite faulty output from Classes I and II.

4.2 Threshold Detection

In the previous analysis, both *operands* for the *relational operators* were *signals/variables*. When one *operand* is a *constant*, the behavior changes to that of a *unary operator* or threshold detector. Treated individually, each threshold value represents a separate function with its own set of generalized faults. The analysis presented here examines those faults as a whole in order to identify patterns in the required test vectors.

4.2.1 Greater Than Signed Threshold

For this development, a signed comparison of a 4-bit number was chosen. The $2^4 = 16$ threshold values provide enough data to identify patterns in the test vectors. Use of 2's complement numbers helps demonstrate the general applicability of the functional analysis techniques.

A 4-bit 2's complement number represents the integer range from -8 to +7. Each threshold function ($A > -8$, $A > -7$, ..., $A > 7$) was analyzed (*SOP* and *POS*) to produce a generalized set of functional faults. A subset of these results is presented in Figure 4-5.

Each threshold is indicated by a heavy vertical line, while the functional test vectors are shaded.

A > -5	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > -4	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > -3	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > -2	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > -1	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > 0	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > +1	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > +2	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7
A > +3	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7

Figure 4-5 Functional test vectors for signed GT threshold.

From these test vectors it is possible to identify a pattern which will produce a behavioral fault model for the threshold functions. Every function requires the two test vectors bracketing the threshold. Each pattern of test vectors is then based on the binary value of the distance the threshold function is from the center of the range of values, indicated by the double lines. Additional test vectors are determined by moving left and right of the initial two test vectors at step sizes starting with the location of the 1s in a binary representation of the distance from center and increasing by powers of 2. The pattern is symmetrical about the threshold, but truncated beyond the far side of the center line.

For example, the threshold for $A > -2$ requires the test vectors -2 and -1 and is one space from the center. Therefore, the additional test vectors to the left of the threshold are $-2 - 1 = -3$ and $-3 - 2 = -5$. To the right of the threshold produces $-1 + 1 = 0$, with the pattern truncated beyond. For a threshold of $A > 4$, the initial test vectors are +4 and +5. The threshold is five spaces from center indicating step sizes of one and four. Thus, additional test vectors to the left are $4 - 1 = 3$ and $3 - 4 = -1$. The only test vector to the right is $5 + 1 = 6$, because $6 + 4 = 12$ is beyond the range of a 4-bit 2's complement number.

4.2.2 A Quick Example

A quick example will be used to demonstrate application of the behavioral fault patterns for threshold functions. Example LE5 in Figure 4-6 defines the behavior of a signed

less than or equal to (LE) threshold function. The integer range -16 to +15 will be synthesized as a 5-bit 2's complement number.

```

entity LE5 is
    port(A: in INTEGER range -16 to +15; LE: out BIT);
end LE5;

architecture BEHAVE of LE5 is
begin
    process(A)
        begin
            if A <= 5 then
                LE <= '1';
            else
                LE <= '0';
            end if;
        end process;
    end BEHAVE;

```

Figure 4-6 Behavioral description for example LE5.

The *LE* function places the threshold between +5 and +6. The threshold lies six spaces from the center of the range of values, implying step sizes of two and four. Therefore, additional test vectors to the left are $5 - 2 = 3$ and $3 - 4 = -1$. Test vectors to the right are $6 + 2 = 8$ and $8 + 4 = 12$. The test vector pattern is represented graphically in Figure 4-7.

A <= 5	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
--------	----	---	---	---	---	---	---	---	---	---	---	----	----	----	----

Figure 4-7 Behavioral test vectors for example LE5.

Example LE5 was synthesized to the gate level circuit shown in Figure 4-8. Fault simulations using the behavioral test vectors from Figure 4-7 resulted in a *SSL* gate level fault coverage of $18/18 = 100\%$.

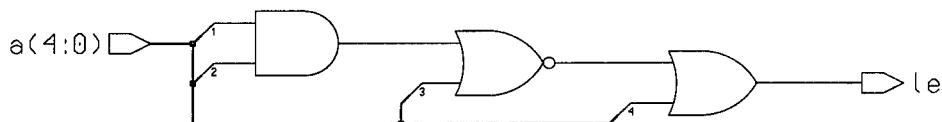


Figure 4-8 Synthesized circuit for example LE5.

The behavioral test vector patterns presented here are equally applicable to unsigned threshold comparisons. For more examples using comparison functions see Appendix A.

4.3 Equal (EQ)

The development of functional and behavioral faults for the *EQ* (and *NE*) function(s) follows the same process as that for *GT*. In addition, since *EQ* is often used in conjunction with other *relational operators*, the function is sometimes formed by a combination of the outputs of the other comparison modules.

4.3.1 Functional Faults

As was the case with the *GT* function, the patterns for the *EQ* function are somewhat vague in the 2-bit case. Therefore, the functional faults and classes will be directly presented for the 3-bit case and then generalized for n-bits. A functional analysis of gate level implementations of the *EQ* function yields the fault classes in Figure 4-9 (*A* and *B* shown in octal).

		A							
		0	1	2	3	4	5	6	7
B	0	I	III	III		III			
	1	II							
	2	II							
	3								
	4	II							
	5								
	6								
	7								I

Figure 4-9 Fault classes for 3-bit EQ function.

Class I faults are again defined along the diagonal of the table, however, only two test vectors are required (00 and 77). Class II faults are selected to be those below the diagonal, while Class III faults are above. Both Class II and III fault sets are of size 3 (*n*).

The patterns apparent in the 3-bit case imply the definition of the functional faults in Table 4-7. The test vectors for Class I faults consist of the all 0's and all 1's cases. Both

Class II and III faults have test vectors where one *operand* is 0 and the other has a single 1, hence the n possible combinations.

Class	Faulty Output	A vs. B	# Test Vectors
I	FALSE	$A = B = 0$ $A = B = 2^n - 1$	2
II	TRUE	$A = 0, B_i = 1$ $i = 0, 1, \dots, n-1$	n
III	TRUE	$A_i = 1, B = 0$ $i = 0, 1, \dots, n-1$	n

Table 4-7 Functional faults for n-bit EQ.

4.3.2 Behavioral Fault Model

The functional faults developed for the *EQ* function can now be abstracted into the behavioral domain, just like those for *GT*. For the Class I faults, the faulty output of FALSE implies the use of an AND/NAND combination. Likewise, the Class II and III faults employ the OR/AND structure seen in the other fault models. The resulting behavioral faults for a 2-bit *EQ* function are presented in Table 4-8.

Class	Faulty Expression	Faults
I	$(A = B) \text{ AND } ((A = "00") \text{ NAND } (B = "00"))$	$A, B = "00"$ $A, B = "11"$
II	$(A = B) \text{ OR } ((A = "00") \text{ AND } (B = "01"))$	$B = "01"$ $B = "10"$
III	$(A = B) \text{ OR } ((A = "01") \text{ AND } (B = "00"))$	$A = "01"$ $A = "10"$

Table 4-8 Behavioral faults for 2-bit EQ.

The functional and behavioral faults developed for the *EQ* function can now be readily adapted to *NE*. Due to the symmetry of the functions, the location of all the test vectors remains the same. Only the reversal of the faulty outputs causes a change in the behavioral fault model.

4.4 Comparison with Previous Fault Models

In contrast to *if-then-else* and *case*, few previous fault models address *relational operators*. As part of the development of the “B-algorithm: A Behavioral Test Generation Algorithm,” [21][22] Cho and Armstrong developed a new behavioral fault model which included *relational operators* as part of *micro-operation* faults. Such faults perturb a *relational operator* to another operator as indicated in Table 4-9. Recall that this fault model suffered from what the authors called the “big micro-operation problem,” where only a single test vector is generated for a complex block of logic. In order to raise their equivalent gate level coverage numbers to acceptable levels, an additional $4n-1$ test vectors are generated by a heuristic test generator for each n -bit micro-operation.

Fault-free Operator	Faulty Operator
BVLT	BVGE
BVLE	BVGT
BVEQ	BVNEQ
BVNEQ	BVEQ

Table 4-9 Micro-operation Faults

A similar fault model was proposed by Al Hayek and Robach [4] as part of a mutation-based testing strategy in which VHDL behavioral faults are considered as software faults. For Relational Operator Replacement (*ROR*), each operator ($<$, $>$, \leq , \geq , $=$, $/=$) is replaced by each of the other relational operators. Mutation analysis does not take into account the size of the hardware implementation, because it considers each relational operator as a software operation and consequently generates only one test vector. In order to improve the performance of their technique, the authors also resorted to heuristics to generate additional test vectors for their complex operators.

The new behavioral fault models developed for *relational operators* are based on the size of the hardware implementation and, therefore, eliminate the need to supplement test vector sets. While the new fault models are definitely more complex than previous ones, this is because they more accurately reflect the underlying complexity of the hardware faults which they attempt to model.

4.5 Application of the New Fault Models

A simple example is now presented to demonstrate the application of the new fault models developed for *relational operators*. A test vector set will be formed based on behavioral faults and then applied to synthesized gate level realizations. Gate level fault coverage of the behavioral test vectors will be used to evaluate the effectiveness of the new behavioral fault models.

Example COMPARE in Figure 4-10 uses two 2-bit signals (A, B) to control the selection of input signals (Y_2, Y_1, Y_0). Based on the relative magnitudes of the control signals, a single input signal is assigned to the output signal (Z). Test vectors can be determined for behavioral faults on the *relational operators* and the control construct *if-then-else*.

```

if A > B then
    Z <= Y2;
elseif A < B then
    Z <= Y0;
else
    Z <= Y1;
end if;

```

Figure 4-10 Behavioral description for example COMPARE.

4.5.1 Faults on Relational Operators

The two *relational operators* ($>$, $<$) in example COMPARE are affected by the three classes of behavioral faults developed earlier in this chapter. Application of each of these fault classes to each of the *relational operators* will determine an appropriate set of behavioral test vectors.

The first *relational operator* in the *expression* $A > B$ controls the *then* clause of the *if-then-else* statement. According to the new behavioral fault model, a Class I fault occurs when the relation $A > B$ produces an erroneous *TRUE* when $A = B$. This fault causes the selection of the $A > B$ clause instead of the desired $A = B$ clause. The test vectors necessary to detect these Class I faults are shown in Table 4-10. For ease of reference when determining compatible fault sets, the faults have been numbered.

Next, a Class II fault for the relation $A > B$ also produces an erroneous *TRUE*, when $A + 1 = B$ (A odd). This fault causes the selection of the $A > B$ clause instead of the

desired $A < B$ clause. Test vectors for the single Class II fault are shown in the middle of Table 4-10.

Finally, a Class III fault produces an erroneous *FALSE* when $A = B + 1$. This fault causes the selection of clauses corresponding to $A \leq B$ instead of the desired $A > B$. Note that the faulty selection of either the $A < B$ clause or the $A = B$ clause results in the elimination of don't care values seen in previous test vectors. The Class III faults and their corresponding test vectors are shown at the bottom of Table 4-10.

Fault Class	Fault #	A	B	(A>B) Y2	(A=B) Y1	(A<B) Y0	Fault-free Z
Class I	1	00	00	1	0	X	0
				0	1	X	1
	2	01	01	1	0	X	0
				0	1	X	1
	3	10	10	1	0	X	0
				0	1	X	1
	4	11	11	1	0	X	0
				0	1	X	1
Class II	5	01	10	1	X	0	0
				0	X	1	1
Class III	6	01	00	0	1	1	0
				1	0	0	1
	7	10	01	0	1	1	0
				1	0	0	1
	8	11	10	0	1	1	0
				1	0	0	1

Table 4-10 Test vectors for behavioral faults for $A > B$.

A similar application of Class I, II, and III faults to the relation $A < B$ in the *elsif* clause produces the test vectors shown in Table 4-11.

Fault Class	Fault #	A	B	(A>B) Y2	(A=B) Y1	(A<B) Y0	Fault-free Z
Class I	9	00	00	X	0	1	0
				X	1	0	1
	10	01	01	X	0	1	0
				X	1	0	1
	11	10	10	X	0	1	0
				X	1	0	1
	12	11	11	X	0	1	0
				X	1	0	1
Class II	13	10	01	0	X	1	0
				1	X	0	1
Class III	14	00	01	1	1	0	0
				0	0	1	1
	15	01	10	1	1	0	0
				0	0	1	1
	16	10	11	1	1	0	0
				0	0	1	1

Table 4-11 Test vectors for behavioral faults for A < B.

Each behavioral fault still has two possible test vectors which will detect it. Application of the control fault model to the *if-then-else* construct will provide additional guidance on selection of a final set of test vectors.

4.5.2 Control Faults

As was shown in Chapter 3, the control fault model specifies that each clause of an *if-then-else* statement can be affected by two different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. Corruptions are caused by clauses which are *logically adjacent* to the affected clause. In the case of *relational operators*, it has been shown that the “>” operator can fault to “=” (Class I) or “<” (Class II), thus causing the *then* clause in example COMPARE to corrupt the either *else* clause or the *elsif* clause. By establishing adjacency using the appropriate values for *A* and *B*, determined by faults to

the *relational operators*, compatible test vectors can be formed for these *Clause-CORRUPT* faults.

First consider the corruptions to the *then* clause by the *elsif* clause. These corruptions occur when the $A < B$ expression in the *elsif* clause produces an erroneous indication that $A > B$, corresponding to a Class II fault for $A < B$. Table 4-11 shows that this Class II fault, **13**, has only one combination of A and B which will produce the proper adjacency between clauses. Using those values for A and B results in the test vectors shown in Table 4-12 for the first two of the *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)* faults for the *then* clause.

Control Fault	Fault Class	A	B	<i>then</i> Y2	<i>else</i> Y1	<i>elsif</i> Y0	Fault-free Z
THEN-CORRUPT (by ELSIF) (OR)	$A < B$ Class II	10	01	0	X	1	0
THEN-CORRUPT (by ELSIF) (AND)				1	X	0	1
THEN-CORRUPT (by ELSE) (OR)	$A > B$ Class III	11	10	0	1	X	0
THEN-CORRUPT (by ELSE) (AND)		01	00	1	0	X	1

Table 4-12 Test vectors for THEN-CORRUPT faults.

The corruptions of the *then* clause by the *else* clause, correspond to the other combinations from Table 4-10 and Table 4-11 where the *then* clause is activated, Class III faults for $A > B$. Table 4-10 shows that faults **6** through **8** provide three combinations of A and B which will produce the proper adjacency between clauses. To reduce the total number of test vectors using *compatible fault* sets, the vectors for the *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)* faults are chosen with different values for A and B . These test vectors form the remainder of Table 4-12.

The *Clause-CORRUPT* faults for the *elsif* and *else* clauses are formed in a similar manner. For the *elsif* clause, the test vectors must set $A < B$, corresponding to Class II faults for $A > B$ and Class III faults for $A < B$. Finally, the *else* clause corresponds to

$A = B$ and the Class I faults. Test vectors for the remaining *Clause-CORRUPT* faults are, therefore, shown in Table 4-13.

Control Fault	Fault Class	A	B	<i>then</i> Y2	<i>else</i> Y1	<i>elsif</i> Y0	Fault-free Z
ELSIF-CORRUPT (by THEN) (OR)	A > B Class II	01	10	1	X	0	0
ELSIF-CORRUPT (by THEN) (AND)				0	X	1	1
ELSIF-CORRUPT (by ELSE) (OR)	A < B Class III	10	11	X	1	0	0
ELSIF-CORRUPT (by ELSE) (AND)		00	01	X	0	1	1
ELSE-CORRUPT (by THEN) (OR)	A > B Class I	00	00	1	0	X	0
ELSE-CORRUPT (by THEN) (AND)		01	01	0	1	X	1
ELSE-CORRUPT (by ELSIF) (OR)	A < B Class I	10	10	X	0	1	0
ELSE-CORRUPT (by ELSIF) (AND)		11	11	X	1	0	1

Table 4-13 Test vectors for ELSIF-CORRUPT and ELSE-CORRUPT faults.

4.5.3 Final Behavioral Test Vector Set

The behavioral faults for the *relational operators* can now be combined with the control faults to form *compatible fault* sets. Recall that Class I faults occur along the diagonal where $A = B$. Notice that the don't care values in the test vectors for faults 1 through 4 are compatible with the test vectors for faults 9 through 12. Plus, due to the symmetry of the behavioral faults defined for *relational operators*, the Class II faults for the relation $A > B$ are a compatible subset of the Class III faults for the relation $A < B$. Likewise, the Class II faults for $A < B$ are a compatible subset of the Class III faults for $A > B$.

Note that the *THEN-CORRUPT* (by ELSIF) and *ELSIF-CORRUPT* (by THEN) faults dictate that both options of test vectors for faults 13 and 5 be chosen. For each of the other *relational operator* faults, only one of the two possible test vectors is needed. For this

example, the final set of test vectors has been chosen with a pattern of alternating fault free values of $Z = 0$ and $Z = 1$. This pattern can be easily repeated for vectors that are not specified by control faults, as will be the case where A and B are larger than two bits.

Application of this selection pattern to each group of faults for the *relational operators* results in a final behavioral test vector set containing 12 test vectors. The test vectors and their corresponding groups are shown in Table 4-14. Note that other test vector sets of size 12 are possible. The required values of A and B are set by the Class I, II and III faults for the *relational operators*, however several choices are possible for Y_2 , Y_1 , and Y_0 . As long as both a $Z = 0$ and $Z = 1$ option are chosen for each fault class, sufficient coverage will be provided for the *Clause-CORRUPT* control faults.

Group # (Control Fault)	Fault #s	A	B	(A>B) Y_2	(A=B) Y_1	(A<B) Y_0	Z
I (ELSE-CORRUPT)	1, 9	00	00	1	0	1	0
	2, 10	01	01	0	1	0	1
	3, 11	10	10	1	0	1	0
	4, 12	11	11	0	1	0	1
II (THEN-CORRUPT)	6	01	00	1	0	0	1
	7, 13	10	01	0	1	1	0
		10	01	1	0	0	1
	8	11	10	0	1	1	0
III (ELSIF CORRUPT)	14	00	01	0	0	1	1
	5, 15	01	10	1	1	0	0
		01	10	0	0	1	1
	16	10	11	1	1	0	0

Table 4-14 Final behavioral test vector set for example COMPARE.

4.6 Evaluation of Behavioral Test Vectors

The test vectors derived from the behavioral faults for example COMPARE are next applied to several synthesized gate level implementations. SSL fault coverage will be determined and used to judge the effectiveness of the behavioral test vectors.

4.6.1 Gate Level Realizations

The VHDL behavioral description for example COMPARE, from Figure 4-10, was synthesized to a gate level implementation using AutoLogic II. The first structural description was produced using minimal optimization in order to produce the most direct realization of the circuit. The gate level circuit for Structure1 is presented in Figure 4-11. Note that the circuit contains a mix of AND, OR, NOT, NAND, and NOR gates and does not directly relate to any of the circuits analyzed in the development of the behavioral fault models for the *relational operators*.

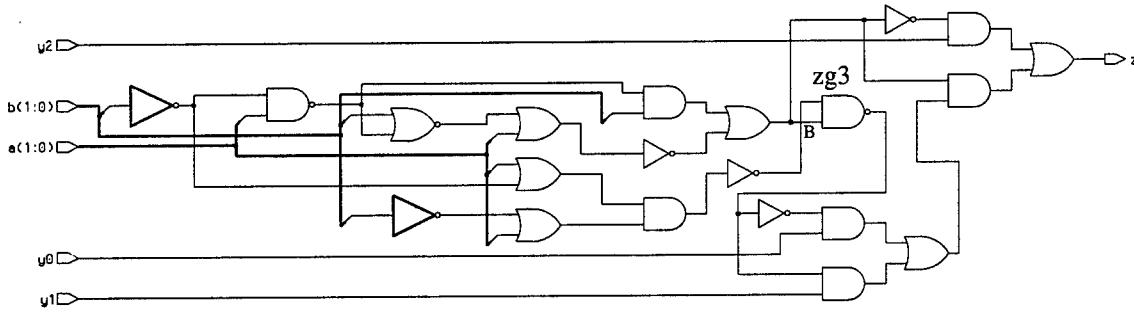


Figure 4-11 Synthesized Structure1 for example COMPARE.

According to MIL-STD 883D, Structure1 contains 74 unique gate level *SSL* faults. Fault simulations were performed using the behavioral test vectors from Table 4-14. The behavioral test vector set detected 73 of the 74 *SSL* gate level faults, resulting in a fault coverage of 98.65%. The undetected fault was on input *B* to NAND gate *zg3* shown in Figure 4-11. Exhaustive testing of the circuit Structure1 reveals that this fault is, in fact, undetectable by any test vector due to redundant logic produced by the synthesis tool. In this case, an alternate measure of effectiveness can also be used to account for redundant logic. The *fault efficiency* is defined as the ratio of detected faults to detectable faults. For this example the fault efficiency is $73/73 = 100\%$.

An alternate test vector set was formed by reversing the order of the $Z = 0$, $Z = 1$ pattern in Table 4-14. Application of this alternate set of behavioral test vectors, shown in Figure 4-12, to the circuit Structure1 produced an identical fault coverage of 98.65%. Either set of test vectors developed from the behavioral fault models, therefore, achieved a *fault efficiency* of 100%.

```
% AB Y2 Y1 Y0 Z : time ;
% Group I
0000 010 1 : 500 ns ;
0101 101 0 : 500 ns ;
1010 010 1 : 500 ns ;
1111 101 0 : 500 ns ;
Group II
0100 011 0 : 500 ns ;
1001 100 1 : 500 ns ;
1001 011 0 : 500 ns ;
1110 100 1 : 500 ns ;
% Group III
0001 110 0 : 500 ns ;
0110 001 1 : 500 ns ;
0110 110 0 : 500 ns ;
1011 001 1 : 500 ns ;
```

Figure 4-12 Alternate set of behavioral test vectors for example COMPARE.

A second gate level implementation of example COMPARE was produced by allowing AutoLogic II to perform logic optimizations. Using these optimizations in a synthesis environment allows a designer to remove redundancies and reduce the number of undetectable faults. The resulting Structure2 is shown in Figure 4-13.

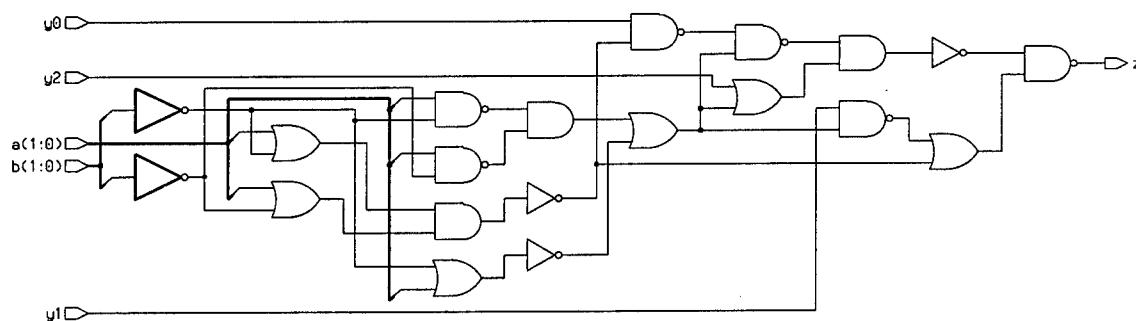


Figure 4-13 Synthesized Structure2 for example COMPARE.

Structure2 contains 72 unique *SSL* gate level faults. Fault simulations using the original and alternate behavioral test vectors from Table 4-14 and Figure 4-12 both result in a fault coverage of $72/72 = 100\%$. The fault coverage as a function of the alternate behavioral test vectors is shown in Figure 4-14.

When redundancies are not present in the gate level circuit, the test vector sets developed using the behavioral fault models achieve a gate level *SSL* fault coverage of 100%.

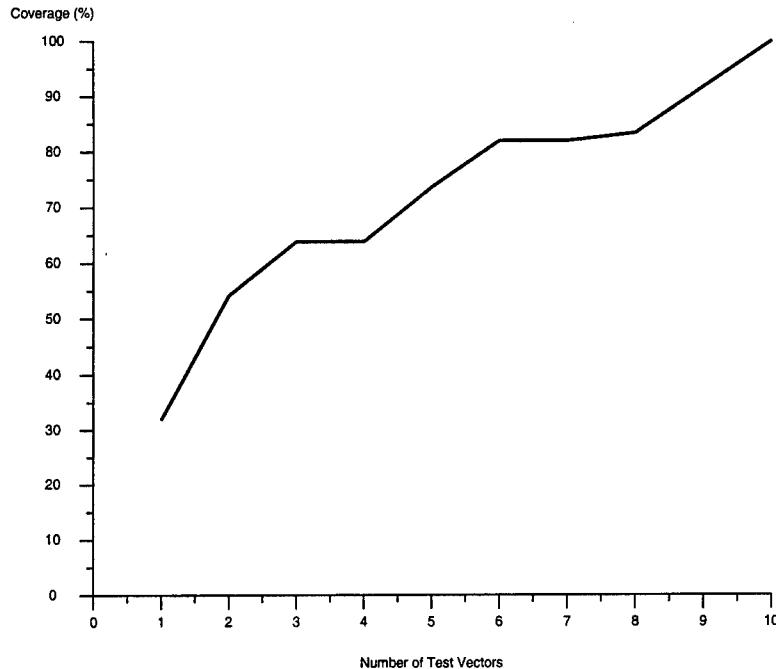


Figure 4-14 Fault coverage for Structure2 of example COMPARE.

These results help validate the new behavioral fault models through practical application. The effects on fault coverage due to expansion of the control signals and data path will now be investigated.

4.6.2 Expansion of the Data Path

In the VHDL behavioral description for example COMPARE, the input signals (Y_2 , Y_1 , Y_0) and the output signal (Z) are each only a single bit wide. To demonstrate the effects of a wider data path on the new behavioral fault models, example COMPARE4 was created by changing the above signals to four bits wide, BIT_VECTOR(3 downto 0). For this example, the control signals A and B remained two bits each.

The faults on the *relational operators* are unchanged due to the widening of the data path. Hence, the values of the control signals A and B are the same as those given in Table 4-14. The only change to the test vectors is the widening of the 1-bit signals to four bits. This is done by replication of the appropriate signal values, since the wider data path simply represents multiple copies of the 1-bit case implemented in parallel.

The control faults on adjacent clauses developed in Section 4.5.2 are, likewise, unaffected by the widening of the data path. The only modifications necessary to the test vec-

tors is again replication of the appropriate signal values. Hence, the behavioral test vectors for example COMPARE4 are presented in Figure 4-15.

```
% AB Y2 Y1 Y0 Z : time ;
% Group I
0000 1111 0000 1111 0000 : 500 ns ;
0101 0000 1111 0000 1111 : 500 ns ;
1010 1111 0000 1111 0000 : 500 ns ;
1111 0000 1111 0000 1111 : 500 ns ;
% Group II
0100 1111 0000 0000 1111 : 500 ns ;
1001 0000 1111 1111 0000 : 500 ns ;
1001 1111 0000 0000 1111 : 500 ns ;
1110 0000 1111 1111 0000 : 500 ns ;
% Group III
0001 0000 0000 1111 1111 : 500 ns ;
0110 1111 1111 0000 0000 : 500 ns ;
0110 0000 0000 1111 1111 : 500 ns ;
1011 1111 1111 0000 0000 : 500 ns ;
```

Figure 4-15 Behavioral test vectors for example COMPARE4.

Example COMPARE4 was synthesized and optimized with AutoLogic II to produce the gate level Structure shown in Figure 4-16. Fault simulations were performed using the test vectors derived from the behavioral fault models. The results show a fault coverage of $150/150 = 100\%$; all SSL gate level faults are detected.

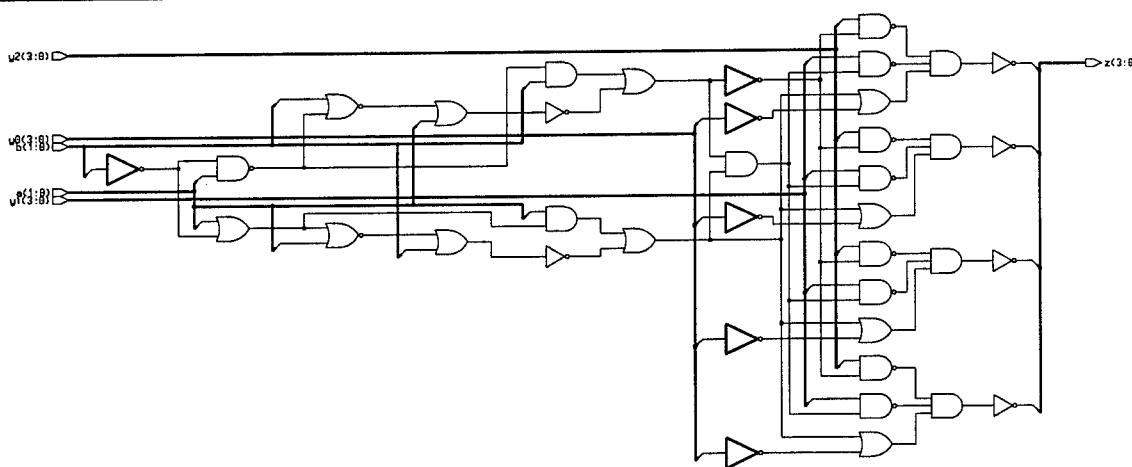


Figure 4-16 Synthesized Structure for example COMPARE4.

As was done previously, an alternate test vector set was formed for example COMPARE4 by reversing the order of the $Z = 0$, $Z = 1$ pattern in Figure 4-15. Fault simulation using these alternate behavioral test vectors again resulted in gate level fault coverage of 100%. Though the number of unique gate level faults more than doubled, the same number of test vectors, 12, were able to provide complete *SSL* fault coverage.

4.6.3 Expansion of the Control Signals

Example COMPARE was next modified by increasing the width of the control signals A and B from two bits to three bits each. The change in size of the control signals does affect the behavioral faults for the *relational operators*. Recall the number of faults on a comparison is related to the number of bits being compared. The increase from two bits to three bits causes the number of Group I faults from Table 4-14 to increase from $2^2 = 4$ to $2^3 = 8$. The number of Group II and Group III faults likewise increases to $2^3 - 1 = 7$.

In contrast to the faults on *relational operators*, the control faults are affected only by adjacency among clauses. Recall from Table 4-13 that the *ELSIF-CORRUPT (by THEN)* faults corresponded to the $A > B$ Class II faults. In the 2-bit case, only a single combination of A and B produced the proper adjacency, forcing the selection of both test vector options to cover both the *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)* faults. In the 3 bit case, there are now $2^{3-1} - 1 = 3$ different combinations of A and B that produce the proper adjacency between clauses (see Figure 4-3). Through proper selection of test vectors from the faults on *relational operators*, it is possible to provide coverage for all control faults without replication of values for A and B . A final set of behavioral test vectors for example COMPARE3 is, hence, presented in Figure 4-17.

```
% A B Y2Y1Y0 Z : time ;
% Group I
000 000 101 0 : 500 ns ;
001 001 101 0 : 500 ns ;
010 010 010 1 : 500 ns ;
011 011 010 1 : 500 ns ;
100 100 101 0 : 500 ns ;
101 101 101 0 : 500 ns ;
110 110 010 1 : 500 ns ;
111 111 010 1 : 500 ns ;
```

Figure 4-17 Behavioral test vectors for example COMPARE3.

```
% Group II
001 000 011 0 : 500 ns ;
010 001 011 0 : 500 ns ;
011 010 100 1 : 500 ns ;
100 011 100 1 : 500 ns ;
101 100 011 0 : 500 ns ;
110 101 011 0 : 500 ns ;
111 110 100 1 : 500 ns ;
% Group III
000 001 110 0 : 500 ns ;
001 010 110 0 : 500 ns ;
010 011 001 1 : 500 ns ;
011 100 001 1 : 500 ns ;
100 101 110 0 : 500 ns ;
101 110 110 0 : 500 ns ;
110 111 001 1 : 500 ns ;
```

Figure 4-17 Behavioral test vectors for example COMPARE3.

Example COMPARE3 was then synthesized and optimized by AutoLogic II to produce the gate level implementation shown in Figure 4-18.

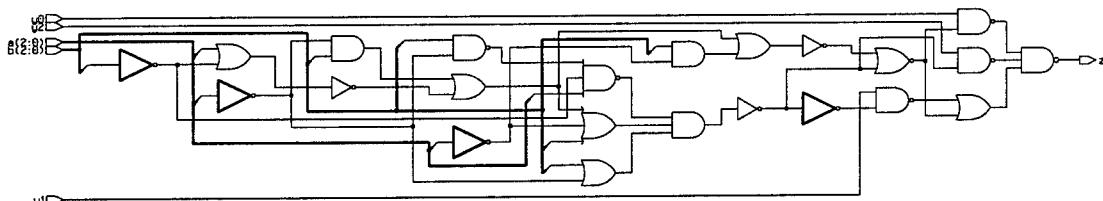


Figure 4-18 Synthesized Structure for example COMPARE3.

Fault simulations were conducted on the gate level Structure for example COMPARE3 using the derived set of behavioral test vectors. A SSL gate level fault coverage of 92/92 = 100% was achieved. As in previous examples, fault simulation with an alternate set of behavioral test vectors again resulted in complete gate level fault coverage.

4.7 Conclusions

New behavioral fault models have been developed for the predefined VHDL *relational operators* from Table 1-2. These fault models are based on a functional analysis of the comparison functions *GT* and *EQ*. The symmetry of these comparison functions allowed the resulting generalized functional faults to be easily adapted for all the *relational operators*.

The new behavioral fault models developed for *relational operators* are based on the size of the hardware implementation and, therefore, eliminate the need to supplement test vector sets via methods such as heuristics. While the new fault models are more complex than previous ones, this is because they more accurately reflect the underlying complexity of the hardware faults which they attempt to model.

A simple example was presented to demonstrate the application of the new fault models. Test vector sets were formed based on behavioral faults to the *relational operators* and the control construct *if-then-else*. These behavioral test vectors were then applied to synthesized gate level realizations. Gate level fault coverage was used to evaluate the effectiveness of the new behavioral fault models.

When redundancies were not present in the synthesized gate level circuits, both the primary and alternate test vector sets developed using the behavioral fault models produced a gate level *SSL* fault coverage of 100%. Even with undetectable faults, the behavioral test vectors were able to achieve a *fault efficiency* of 100%.

Chapter 5

Arithmetic Operators

Like the *relational operators*, *arithmetic operators* also generate large blocks of combinational logic. The predefined VHDL operators *ADD* (+) and *SUB* (-) are normally implemented by synthesis tools with standard library modules. Optimizations for speed or chip area may modify these building blocks, however, the basic function of the *arithmetic operators* remains unchanged. A fault modeling technique is proposed here based on complete functional testing of the arithmetic building blocks. By concentrating on functional testing, complete gate level *SSL* fault coverage should be obtained over a broad range of hardware implementations.

Previous behavioral modeling approaches, based on perturbing language constructs such as *ADD* to *SUB*, do not accurately reflect underlying hardware faults. In order to compensate for this “big micro-operation problem,” alternate methods such as heuristics were used to supplement test vector sets to increase the equivalent gate level fault coverage. The new modeling technique presented in this chapter increases the complexity of the fault models for the *arithmetic operators*, providing a better representation of the faults which occur in actual hardware.

5.1 Addition

The *ADD* operation has several basic forms which will be investigated in succession. A two-level network would be the fastest, however, this circuit would require a large number of gates and gate inputs. It would be necessary to have 2^{2n} NAND gates of $2n + 1$ inputs and one NAND gate of 2^{2n} inputs to add two n -bit numbers [41]. This number of gates and inputs is quite significant for even small values of n .

In contrast to this direct approach, adders are most commonly implemented by the interconnection of smaller functional building blocks. In its simplest form, a *half adder* (*HA*) is a multiple output combinational circuit which adds two bits to produce a *sum* and a *carry-out*. A *full adder* (*FA*) adds two binary digits and a *carry-in* from a previous stage. To speed up the combinational addition process, by reducing the rippling of carries between stages, methods such as *carry look-ahead* (*CLA*) are used.

5.1.1 Ripple Carry Adder

Behavioral modeling of the *ADD* operation has two basic forms, depending on the presence of an overall *carry-in* and *carry-out* for the resulting adder circuit. In its simplest form, the addition of two n -bit binary numbers can be represented as:

$$S \leq A + B;$$

A 3-bit *ripple carry* implementation of this *ADD* operation is shown in Figure 5-1.

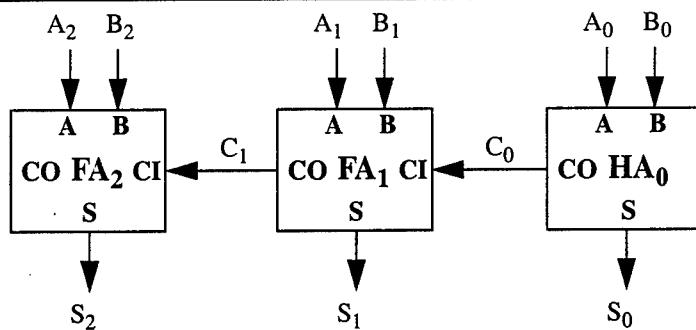


Figure 5-1 Ripple carry adder.

The *full adders* (FA_i) are 3-input 2-output combinational circuits, where *CI* and *CO* represent the *carry-in* and *carry-out* respectively. A truth table for the *full adder* function is presented in Table 5-1; Test # shown in octal. Similarly, the *half adder* (HA_0) is a 2-input 2-output circuit that can be considered a subset of the *full adder* function. The upper half of Table 5-1, where $CI = 0$, represents the truth table for the *half adder* function.

Test #	CI	A	B	S	CO
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

Table 5-1 Truth table for full adder.

5.1.1.1 Functional Testing

A functional testing strategy is presented which will ensure complete gate level SSL fault coverage for a broad range of *ripple carry* adder implementations. Because such realizations are one-dimensional cellular logic arrays, made up of 2- and 3-input functional building blocks, complete functional testing can be achieved by exhaustive testing of each module [44].

Table 5-2 presents the Phase I functional tests proposed for the 3-bit *ripple carry* adder. The left hand side of the table shows the *ADD* operation with the resulting *sum*. Non-zero carries between stages of the adder are shown by arrows above the columns. The right hand side of the table indicates the **Test #**, from Table 5-1, which is applied to each functional module (*FA*, *HA*) by the input test vectors. For example, the first row shows that A_i and B_i are 0 for every stage of the adder. Since the resulting *carry-out* for each stage is also 0, **Test 0** is applied to all modules of the adder.

$\begin{array}{r} A \\ + B \\ \hline S \end{array}$	Test #		
	FA ₂	FA ₁	HA ₀
$\begin{array}{r} 000 \\ + 000 \\ \hline 000 \end{array}$	0	0	0
$\begin{array}{r} 000 \\ + 111 \\ \hline 111 \end{array}$	1	1	1
$\begin{array}{r} 111 \\ + 000 \\ \hline 111 \end{array}$	2	2	2
$\begin{array}{r} 111 \\ + 111 \\ \hline 110 \end{array}$	7	7	3

Table 5-2 Phase I functional tests.

Note that the last row of Table 5-2 shows that both C_0 and C_1 are 1, resulting in **Test 7** being applied to FA_1 and FA_2 . Since HA_0 only has 2 inputs, the last test from Phase I represents **Test 3** for this module and concludes complete functional testing of the *half adder*.

To continue the testing of the *full adder* modules, further vectors must be generated which will produce the remaining input and carry combinations. Consider **Test 5** and **Test 6**, which require opposite inputs for A and B with $CI = 1$. In order to start the *ripple carry* process between stages, HA_0 must generate a $CO = 1$. Since **Test 5** and **Test 6** produce a $CO = 1$, the carries are correctly propagated through the stages. Table 5-3 summarizes the results for the Phase II functional tests.

$\begin{array}{r} A \\ + B \\ \hline S \end{array}$	Test #		
	FA ₂	FA ₁	HA ₀
$\begin{array}{r} 1 \\ 001 \\ + 111 \\ \hline 000 \end{array}$	5	5	3
$\begin{array}{r} 1 \\ 111 \\ + 001 \\ \hline 000 \end{array}$	6	6	3

Table 5-3 Phase II functional tests.

Only **Test 3** and **Test 4** remain to complete the testing of the *full adder* modules. Since **Test 3** requires a $CI = 0$, yet produces a $CO = 1$, no single vector can provide **Test 3** inputs to all stages. Likewise, **Test 4** requires $CI = 1$ and produces $CO = 0$. Due to the observed symmetry, the functional tests for Phase III can be formed by interleaving **Test 3** and **Test 4** as shown in Table 5-4.

$\begin{array}{r} A \\ + B \\ \hline S \end{array}$	Test #		
	FA ₂	FA ₁	HA ₀
$\begin{array}{r} 1 \\ 101 \\ + 101 \\ \hline 010 \end{array}$	3	4	3
$\begin{array}{r} 1 \\ 010 \\ + 010 \\ \hline 100 \end{array}$	4	3	0

Table 5-4 Phase III functional tests.

Hence, complete functional testing of all modules of the *ripple carry* adder in Figure 5-1 has been achieved with eight test vectors. Next, the scalability of the functional tests will be evaluated for larger adders.

5.1.1.2 Scalability

Because the *ripple carry* structure is made up entirely of cascaded HA and FA modules, the functional tests developed in Section 5.1.1.1 can be readily adapted to larger adders. Since the functional tests are based on the complete testing of each individual module, only eight test vectors are required regardless of the size of the inputs *A* and *B* [44]. A 4-bit addition will be used to demonstrate the scaling of the test vectors.

Table 5-5 presents the test vectors for a 4-bit *ripple carry* adder. The tests for Phases I and II are created by replication of the highest order bits of the 3-bit case. The test vectors for Phase III are produced by continuation of the alternating patterns caused by the interleaving of **Test 3** and **Test 4** for the FA modules. Note that both extensions of the functional tests can be continued for larger values of *n*, still requiring only eight test vectors.

Phase	A	B	S	Test #			
				FA ₃	FA ₂	FA ₁	HA ₀
I	0000	0000	0000	0	0	0	0
	0000	1111	1111	1	1	1	1
	1111	0000	1111	2	2	2	2
	1111	1111	1110	7	7	7	3
II	0001	1111	0000	5	5	5	3
	1111	0001	0000	6	6	6	3
III	0101	0101	1010	4	3	4	3
	1010	1010	0100	3	4	3	0

Table 5-5 Functional tests for 4-bit ripple carry adder.

5.1.1.3 Behavioral Fault Model

Comparison of the functional test vectors with the gate level faults detected provides some insight into the performance of the functional testing technique. For example, the test vector *A* = 0000, *B* = 0000 covers faults which correspond to the *sum* and *carry* bits of

each stage producing an erroneous 1. Due to the relative bit positions of the various stages, this causes the resulting *sum* to be in error by either +1, +2, +4, or +8. Similar observations reveal that the functional faults for the *ripple carry* adder result in an output *sum* which is in error by $\pm 2^i$ ($i = 0, 1, 2, 3$).

A one-to-one correspondence cannot be established between the large number of functional faults which are covered by the relatively small number of eight functional test vectors. Therefore, a direct mapping cannot be made to produce a simple behavioral fault model. Rather, the approach taken here simply seeks to map the functional test vectors into error vectors which corrupt the *ADD* operation for the appropriate input combinations.

Again consider the functional test vector $A = 0000$, $B = 0000$. Corruption of the resulting *sum* can be achieved by producing an erroneous 1 in any of the bit positions. For this fault model, the corruptions are chosen to be to the least significant bit position. The *XOR* operator provides the desired corruption properties by inverting the appropriate bit when presented with an error vector of 0001.

A behavioral fault model for the first functional test vector is therefore proposed as:

```
D <= (A + B) XOR "000" & (A = "0000" AND B = "0000")
```

The *concatenation operator* (*&*) combines the *TRUE/FALSE* from the *AND* operator with leading 0's to produce the appropriate error vector. While this is not syntactically correct VHDL due to type differences, it presents the concept of the behavioral fault model. A complete implementation of the behavioral fault model using functions from the Mentor Graphics *std_logic_arith* library is presented in Figure 5-2.

```
D <= (A + B) XOR zero_extend(to_stdlogic(
    (A = "0000" AND B = "0000") OR
    (A = "0000" AND B = "1111") OR
    (A = "1111" AND B = "0000") OR
    (A = "1111" AND B = "1111") OR
    (A = "0001" AND B = "1111") OR
    (A = "1111" AND B = "0001") OR
    (A = "0101" AND B = "0101") OR
    (A = "1010" AND B = "1010")), 4);
```

Figure 5-2 Behavioral fault model for ripple carry adder.

5.1.1.4 Evaluation of the Behavioral Test Vectors

The behavioral test vectors derived in the preceding sections will now be applied to gate level implementations of *ripple carry* adders. Fault simulations will determine *SSL* fault coverage and demonstrate the effectiveness of this functional testing technique.

The VHDL behavioral description for example ADD4 is shown in Figure 5-3. Example ADD4 was synthesized with AutoLogic II to produce the *ripple carry* circuit shown in Figure 5-4.

```

entity add4 is
    port(A, B: in std_logic_vector(3 downto 0);
         D: out std_logic_vector(3 downto 0));
end add4;

architecture behave of add4 is
begin
    process(A, B)
    begin
        D <= A + B;
    end process;
end behave;

```

Figure 5-3 Behavioral description for example ADD4.

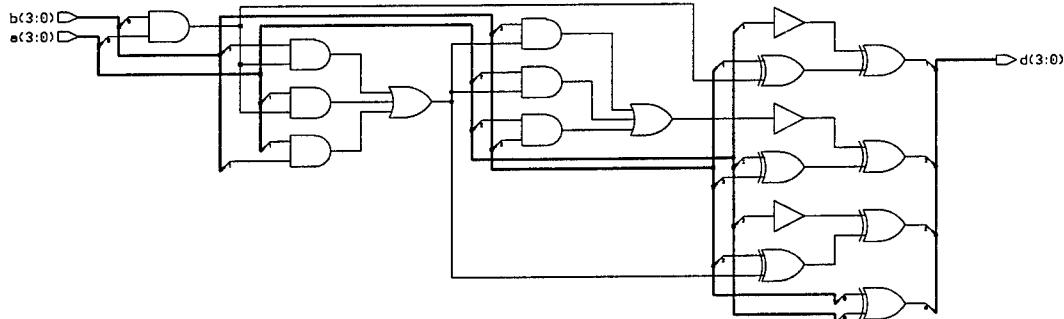


Figure 5-4 Synthesized circuit for example ADD4.

According to MIL-STD 883D, the synthesized circuit contains 102 unique gate level *SSL* faults. Fault simulations using the behavioral test vectors derived from Table 5-5 result in a fault coverage of $102/102 = 100\%$.

An alternate realization of the 4-bit *ripple carry* adder is presented in Figure 5-5. The NOR-only circuit, based on *POS* implementations of the *FA* and *HA* modules, contains

142 unique gate level *SSL* faults. Fault simulations using the eight behavioral test vectors again resulted in complete gate level fault coverage.

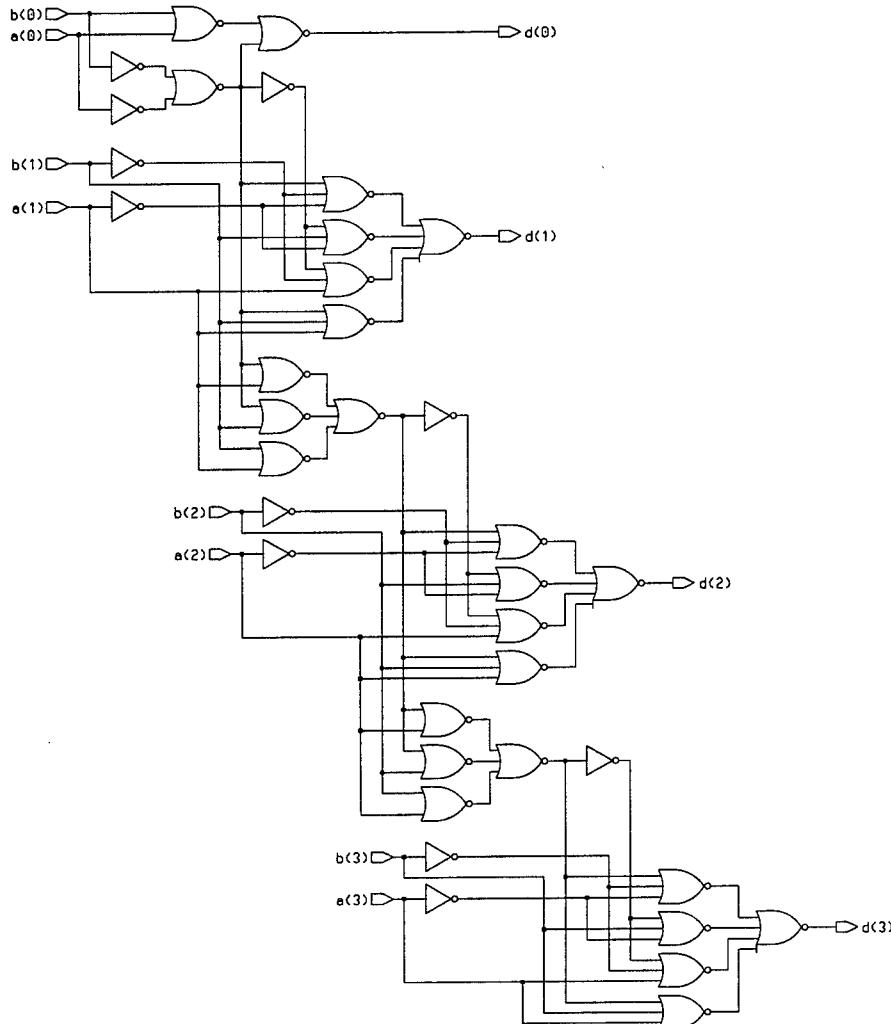


Figure 5-5 NOR-only realization of example ADD4.

5.1.1.5 Carry-in and Carry-out

The more complex form of behavioral addition includes a *carry-in* and/or a *carry-out*. Since the predefined VHDL *ADD* operator combines two n -bit operands to form an n -bit result, some additional manipulation is required to deal with the extra carries. The *carry-out* is produced by simply extending the *ADD* operation to $n+1$ bits and extracting the most significant bit of the result. The *carry-in* can be modeled by an extra addition of a single bit. Example ADD4wc, in Figure 5-6, demonstrates the behavioral description of a 4-bit addition with carries.

```

entity add4wc is
    port(A, B: in std_logic_vector(3 downto 0);
         D: out std_logic_vector(3 downto 0);
         CIN: in std_logic;
         COUT: out std_logic);
end add4wc;

architecture behave of add4wc is
begin
process(A,B,CIN)
    variable op1,op2,sum: std_logic_vector(4 downto 0);
    variable carry_in : std_logic_vector(1 downto 0);
begin
    op1 := '0' & A;
    op2 := '0' & B;
    carry_in := '0' & CIN;
    sum := op1 + op2 + carry_in;
    D <= sum(3 downto 0);
    COUT <= sum(4);
end process;
end behave;

```

Figure 5-6 Behavioral description for example ADD4wc

Rather than treating the two (+) *operators* separately in example ADD4wc, a synthesis tool will combine the *operators* to form a single adder. Recognition of this *carry-in* structure can be used to produce an appropriate behavioral fault model. Recall from the functional testing strategy in Section 5.1.1.1 that the least significant stage of the adder was only a *half-adder*. The only change when a *carry-in* is present is the conversion of this stage to a *full-adder*. Minor adjustments to the test vectors will ensure complete functional testing of this new module.

The test vectors for *A* and *B* remain the same for both Phase I and Phase III tests to the adder. The value of the *CIN* is simply set to match the other carry patterns for that test. For example, the last test for Phase I applies **Test 7** to each *FA*, thus *CIN = 1*. The test vectors for Phase II originally set the least significant bit of *A* or *B* to *1* to initiate the *ripple carries* through the adder. *CIN* now serves as the least significant input the adder and can assume that role. The resulting functional test vectors are presented in Table 5-6.

The behavioral fault model for the *add with carry* operation follows the same approach shown in Section 5.1.1.3. The functional test vectors are converted to error vectors which

Phase	A	B	CIN	COUT	S	Test #			
						FA ₃	FA ₂	FA ₁	FA ₀
I	0000	0000	0	0	0000	0	0	0	0
	0000	1111	0	0	1111	1	1	1	1
	1111	0000	0	0	1111	2	2	2	2
	1111	1111	1	1	1111	7	7	7	7
II	0000	1111	1	1	0000	5	5	5	5
	1111	0000	1	1	0000	6	6	6	6
III	0101	0101	0	0	1010	4	3	4	3
	1010	1010	1	1	0101	3	4	3	4

Table 5-6 Functional tests for example ADD4wc.

corrupt the resulting sum. Application of the eight behavioral test vectors derived from Table 5-6, to a synthesized gate level implementation of example ADD4wc, results in the expected complete gate level fault coverage.

5.1.2 Carry Look-Ahead Adder

Carry look-ahead (CLA) speeds up the process of combinational addition by determining carries for higher order stages of the adder without having to wait for them to ripple through lower order stages. From the truth table for the FA, Table 5-1, it can be seen that the *carry-out* is the same as the *carry-in* as long as one of the other inputs is a 1. Also, the *carry-out* is always a 1 independent of the *carry-in* when both of the other inputs are 1s, and a 0 if both are 0. Consequently, two useful functions can be defined: the *carry-propagate*, P_i , and the *carry-generate*, G_i [41].

$$\begin{aligned} P_i &= A_i \oplus B_i \\ G_i &= A_i B_i \end{aligned}$$

The FA equations can then be written as:

$$\begin{aligned} S_i &= P_i \oplus C_{i-1} \\ C_i &= G_i + P_i C_{i-1} \end{aligned}$$

A *CLA* realization of a 3-bit adder is shown in Figure 5-7, where PG_i represents *propagate-generate* modules and SU_i implements the *sum* functions.

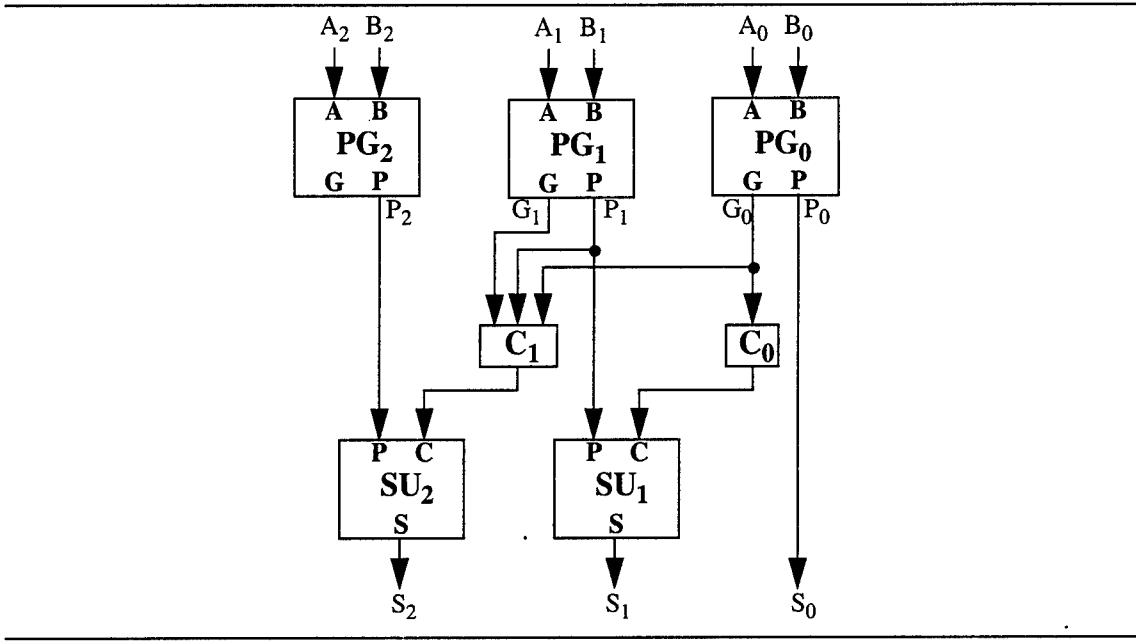


Figure 5-7 Carry look-ahead adder.

The carries for the various stages of a 4-bit adder are formed from the terms shown in Table 5-7. The columns indicate the contributions of the various stages, which are combined to produce the appropriate carry. Presence or absence of a *carry-in* (C_{-1}), to the least significant stage of the adder, determines whether or not to use the terms in the farthest right column.

Carry	Stage				
	3	2	1	0	-1
C_0				G_0	P_0C_{-1}
C_1			G_1	P_1G_0	$P_1P_0C_{-1}$
C_2		G_2	P_2G_1	$P_2P_1G_0$	$P_2P_1P_0C_{-1}$
C_3	G_3	P_3G_2	$P_3P_2G_1$	$P_3P_2P_1G_0$	$P_3P_2P_1P_0C_{-1}$

Table 5-7 Carries for 4-bit CLA adder.

5.1.2.1 Functional Testing

Carry-propagate and *carry-generate* are both 2-input functions which will be completely tested by the functional test vectors developed for the *ripple carry* adder. The *sum* (S_i) is still a function of A_i , B_i , and C_{i-1} and, therefore, will also be exhaustively tested by the *ripple carry* test vectors. Examination of the carries from Table 5-7 indicates that any

faults to lower order carries will *dominate* faults in higher order carries. Hence, testing for all the faults in the highest order carry in a *CLA* adder will provide coverage for all lower order faults.

A simple 4-bit addition will be used to evaluate the coverage of carry faults by the *ripple carry* test vectors. Since this circuit has neither a *carry-in* nor a *carry-out*, the highest order carry is C_2 , which contains three terms: G_2 , P_2G_1 , and $P_2P_1G_0$. Possible functional faults include a *missing carry*, due to one of the terms producing an erroneous 0, and an *extra carry*, due to a term producing an unwanted 1.

Missing carry faults are evaluated in Table 5-8. In order to establish test vectors, G_i is set to 1 by $A_i = B_i = 1$ and P_i is set to 1 by $A_i = \bar{B}_i$. The right hand column indicates whether or not the indicated fault is covered by a *ripple carry* functional test vector.

Stage	Term	Test Set-up	Test Vector	Covered
0	$P_2P_1G_0$	$ \begin{array}{r} X \ 0 \ 0 \ 1 \\ + X \ 1 \ 1 \ 1 \\ \hline P_2P_1G_0 \end{array} $	$ \begin{array}{r} 0001 \\ + 1111 \\ \hline 0000 \end{array} $	Yes
1	P_2G_1	$ \begin{array}{r} X \ 0 \ 1 \ X \\ + X \ 1 \ 1 \ X \\ \hline P_2G_1 \end{array} $	$ \begin{array}{r} 0010 \\ + 1111 \\ \hline 0001 \end{array} $	No
2	G_2	$ \begin{array}{r} X \ 1 \ X \ X \\ + X \ 1 \ X \ X \\ \hline G_2 \end{array} $	$ \begin{array}{r} 1111 \\ + 1111 \\ \hline 1110 \end{array} $	Yes

Table 5-8 Missing carry faults.

The *missing carry* fault for stage 1 is not covered by the functional test vectors for the *ripple carry* adder. Examination of larger *carry look-ahead* circuits reveals that only the *missing carry* faults for the most and least significant stages will be covered. Hence, an additional $n - 3$ test vectors will be required to provide complete fault coverage.

Extra carry faults are evaluated in Table 5-9. An erroneous 1 in a term can be caused by any element in that term producing a 1 when it has been set to 0. Two *extra carry* faults are not covered by the functional test vectors for the 4-bit *ripple carry* adder. All of the erroneous generation faults (G_i) are covered as well as the erroneous propagation fault for the highest order *carry-propagate* (P_2).

Stage	Term	Test Set-up	Test Vector	Covered
0	$P_2 P_1 G_0$	$\begin{array}{r} X \ 0 \ 0 \ 0 \\ + X \ 1 \ 1 \ 1 \\ \hline P_2 P_1 \bar{G}_0 \end{array}$	$\begin{array}{r} 0000 \\ + 1111 \\ \hline 1111 \end{array}$	Yes
		$\begin{array}{r} X \ 0 \ 0 \ 1 \\ + X \ 1 \ 0 \ 1 \\ \hline P_2 \bar{P}_1 G_0 \end{array}$	$\begin{array}{r} 0001 \\ + 1101 \\ \hline 1110 \end{array}$	No
		$\begin{array}{r} X \ 0 \ 0 \ 1 \\ + X \ 0 \ 1 \ 1 \\ \hline \bar{P}_2 P_1 G_0 \end{array}$	$\begin{array}{r} 0001 \\ + 1011 \\ \hline 1100 \end{array}$	No
1	$P_2 G_1$	$\begin{array}{r} X \ 0 \ 0 \ X \\ + X \ 1 \ 1 \ X \\ \hline P_2 \bar{G}_1 \end{array}$	$\begin{array}{r} 0000 \\ + 1111 \\ \hline 1111 \end{array}$	Yes
		$\begin{array}{r} X \ 0 \ 1 \ X \\ + X \ 0 \ 1 \ X \\ \hline \bar{P}_2 G_1 \end{array}$	$\begin{array}{r} 1010 \\ + 1010 \\ \hline 0100 \end{array}$	Yes
2	G_2	$\begin{array}{r} X \ 0 \ X \ X \\ + X \ 1 \ X \ X \\ \hline \bar{G}_2 \end{array}$	$\begin{array}{r} 0000 \\ + 1111 \\ \hline 1111 \end{array}$	Yes

Table 5-9 Extra carry faults.

The behavioral fault model for the *CLA* adder follows directly from the base *ripple carry* fault model. Additional functional test vectors for *missing carry* and *extra carry* faults are also mapped into error vectors which corrupt the resulting sum. The behavioral test vectors, therefore, consist of the eight *ripple carry* tests supplemented by some number of *CLA* tests. The additional behavioral test vectors for a 4-bit *CLA* adder are summarized in Table 5-10.

Fault	Stage	A	B	S
Missing Carry	1	0010	1111	0001
Extra Carry	0	0001	1101	1110
		0001	1011	1100

Table 5-10 Additional behavioral test vectors for CLA adder.

5.1.2.2 Application of the Behavioral Test Vectors

Behavioral test vectors are now applied to a *CLA* implementation of example ADD4. The NAND-only circuit in Figure 5-8 contains 130 unique *SSL* gate level faults. Fault simulation using the original eight *ripple carry* test vectors from Table 5-5 produces a gate level fault coverage of $127/130 = 97.69\%$. Examination of the results confirms that the three uncovered faults are, in fact, from signals forming the highest order carry, C_2 . Application of the additional *CLA* test vectors from Table 5-10 then achieves complete fault coverage.

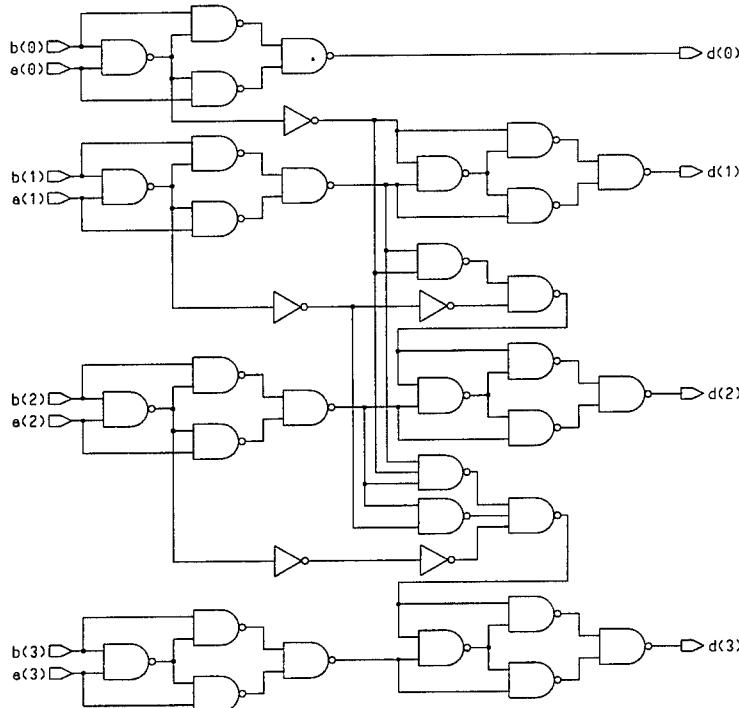


Figure 5-8 CLA implementation of example ADD4.

5.1.2.3 Scalability

Gate level fault coverage can now be evaluated for a larger example, ADD8. Fault simulations using behavioral test vectors will be performed for both *ripple carry* and *CLA* implementations. Optimizations can also be investigated when detailed knowledge of the target technology is available.

Example ADD8 was synthesized to a gate level circuit producing the *ripple carry* implementation as shown in Figure 5-9. Fault simulations with eight behavioral test vec-

tors, extrapolated from example ADD4, produce a SSL gate level fault coverage of $234/234 = 100\%$. The fault coverage plot is shown in Figure 5-10.

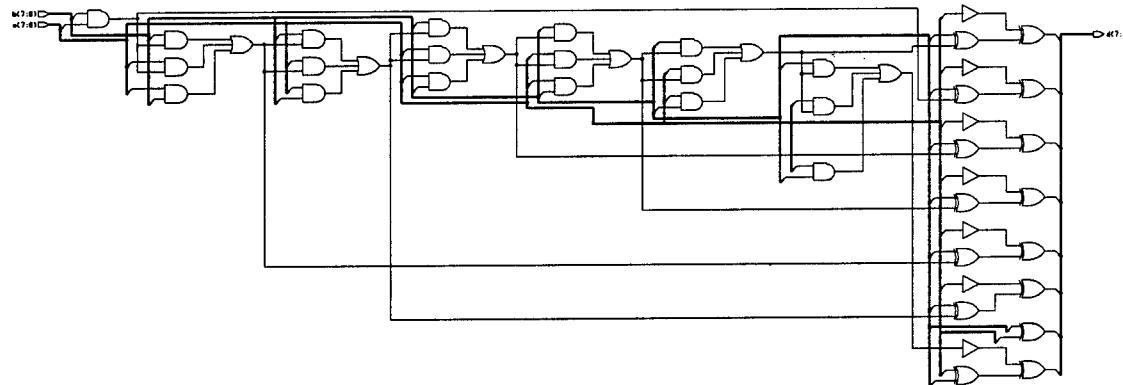


Figure 5-9 Ripple carry implementation of example ADD8.

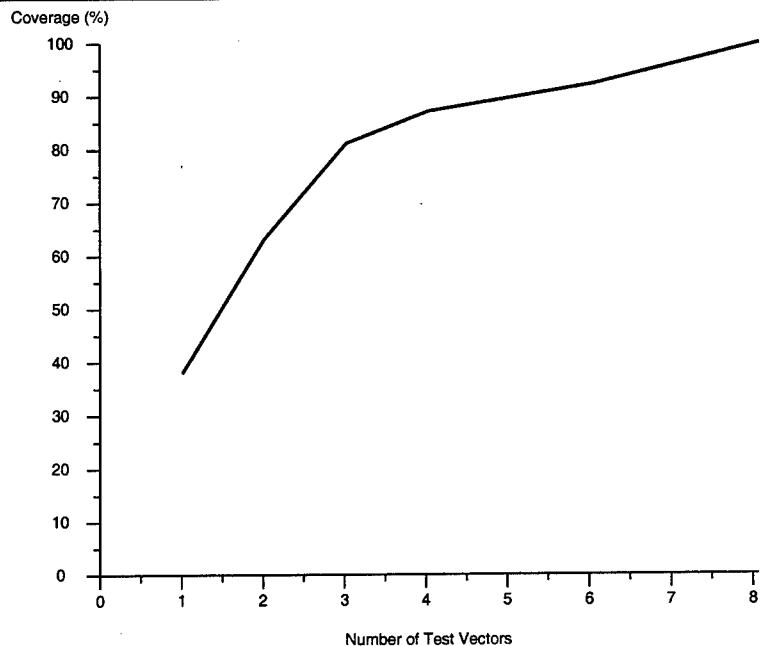


Figure 5-10 Fault coverage for ripple carry ADD8.

Next a *CLA* implementation of example ADD8 will be considered. Without specific knowledge of the details of the target technology, behavioral test vectors are first formed on the basis of a full 8-bit *CLA* structure.

The highest order carry in an 8-bit *CLA* adder, without a *carry-in* or *carry-out*, is C_6 . Extrapolating from Table 5-7, the terms for C_6 are $P_6P_5P_4P_3P_2P_1G_0$, $P_6P_5P_4P_3P_2G_1$, $P_6P_5P_4P_3G_2$, $P_6P_5P_4G_3$, $P_6P_5G_4$, P_6G_5 , and G_6 . According to the new behavioral fault

model, test vectors for *extra carry* faults are required for all but the two most significant stages. Each stage has a separate fault for each *carry-propagate* (P_i) signal in that term. Behavioral test vectors for the *extra carry* faults are presented in Table 5-11. Test vectors are necessary for *missing carry* faults for all but the lowest and highest order terms. The *missing carry* behavioral test vectors are presented in Table 5-12.

Stage	Term	A	B	S
0	$P_6P_5P_4P_3P_2P_1G_0$	00000001	11111101	11111110
		00000001	11111011	11111100
		00000001	11110111	11111000
		00000001	11101111	11110000
		00000001	11011111	11100000
		00000001	10111111	11000000
1	$P_6P_5P_4P_3P_2G_1$	00000010	11111011	11111101
		00000010	11110111	11111001
		00000010	11101111	11110001
		00000010	11011111	11100001
		00000010	10111111	11000001
2	$P_6P_5P_4P_3G_2$	00000100	11110111	11111011
		00000100	11101111	11110011
		00000100	11011111	11100011
		00000100	10111111	11000011
3	$P_6P_5P_4G_3$	00001000	11101111	11110111
		00001000	11011111	11100111
		00001000	10111111	11000111
4	$P_6P_5G_4$	00010000	11011111	11101111
		00010000	10111111	11001111

Table 5-11 Behavioral test vectors for extra carry faults.

Stage	Term	A	B	S
1	$P_6P_5P_4P_3P_2G_1$	00000010	11111111	00000001
2	$P_6P_5P_4P_3G_2$	00000100	11111111	00000011
3	$P_6P_5P_4G_3$	00001000	11111111	00000111
4	$P_6P_5G_4$	00010000	11111111	00001111
5	P_6G_5	00100000	11111111	00011111

Table 5-12 Behavioral test vectors for missing carry faults.

A common modular *CLA* implementation was chosen, which cascades individual 4-bit *CLA* adders to form the n -bit addition. A block diagram of an 8-bit adder, therefore, has the structure shown in Figure 5-11.

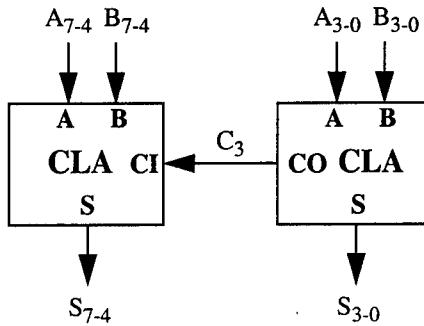


Figure 5-11 Block diagram of modular CLA adder.

Fault simulations were performed on a NAND-only realization of the modular *CLA* adder using the combined behavioral test vectors. The original eight *ripple carry* test vectors produce a *SSL* gate level fault coverage of $290/310 = 93.55\%$. Application of the additional 25 *carry look-ahead* test vectors results in the expected complete gate level fault coverage.

Note the relative inefficiency of the *CLA* test vectors versus the original eight *ripple carry* vectors in the fault coverage plot in Figure 5-12. In addition, there is a large flat portion of the graph, between vectors 21 to 31, where fault coverage does not improve. This is due to the lack of specific knowledge about the modular *CLA* implementation. A designer equipped with details of the functional elements used in the target technology can optimize the behavioral test vectors.

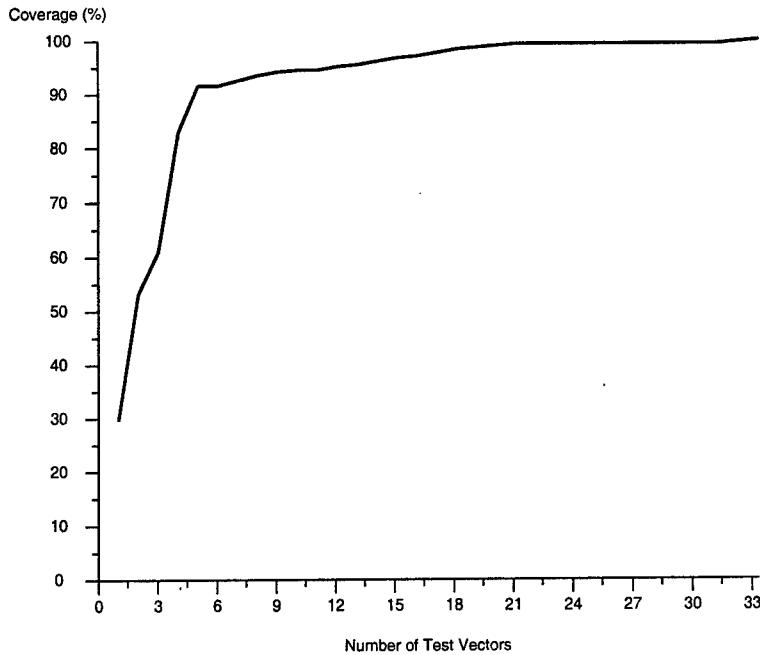


Figure 5-12 Fault coverage for modular CLA adder.

5.1.2.4 Optimization of CLA Behavioral Faults

Given the additional detail that adders in a certain design are implemented by cascading 4-bit *CLA* modules, an optimized set of behavioral test vectors can be derived. From the block diagram of the modular *CLA* adder in Figure 5-11, highest order carries can be determined for both the upper, S_{7-4} , and lower, S_{3-0} , *CLA* modules.

The highest order carry in the lower *CLA* adder, with a *carry-out* and no *carry-in*, is C_3 . From Table 5-7, the terms for C_3 are $P_3P_2P_1G_0$, $P_3P_2G_1$, P_3G_2 , and G_3 . The highest order carry in the upper *CLA* adder, with a *carry-in* and no *carry-out*, is C_6 . Extrapolating from Table 5-7, the terms for C_6 are $P_6P_5P_4C_3$, $P_6P_5G_4$, P_6G_5 , and G_6 .

According to the new behavioral fault model, test vectors are necessary for *missing carry* faults for all but the lowest and highest order terms for each *CLA* module. The *missing carry* behavioral test vectors are, therefore, presented in Table 5-13. Test vectors for *extra carry* faults are required for all but the two most significant stages for each module. Each stage has a separate fault for each *carry-propagate* (P_i). Behavioral test vectors for the *extra carry* faults are shown in Table 5-14.

Fault simulations were performed on the modular *CLA* adder using the optimized behavioral test vectors from Table 5-13 and Table 5-14. A *SSL* gate level fault coverage of

Stage	Term	A	B	S
1	$P_3P_2G_1$	00000010	11111111	00000001
2	P_3G_2	00000100	11111111	00000011
4	$P_6P_5G_4$	00010000	11111111	00001111
5	P_6G_5	00100000	11111111	00011111

Table 5-13 Optimized test vectors for missing carry faults.

Stage	Term	A	B	S
0	$P_3P_2P_1G_0$	00000001	11111101	11111110
		00000001	11111011	11111100
		00000001	11110111	11111000
1	$P_3P_2G_1$	00000010	11111011	11111101
		00000010	11110111	11111001
3	$P_6P_5P_4C_3$	00001000	11101111	11110111
		00001000	11011111	11100111
		00001000	10111111	11000111
4	$P_6P_5G_4$	00010000	11011111	11101111
		00010000	10111111	11001111

Table 5-14 Optimized test vectors for extra carry faults.

$310/310 = 100\%$ was achieved with only $8 + 14 = 22$ behavioral test vectors as opposed to the $8 + 25 = 33$ non-optimized vectors. Hence, additional knowledge about the target architecture has allowed optimization of the behavioral test vector set while still achieving complete gate level fault coverage.

5.1.3 Summary

A behavioral fault model for addition has been derived using a complete functional testing technique. Both simple addition and the more complex form including a *carry-in* and/or a *carry-out* have been considered. The functional tests can be readily extended to n -bits, still requiring only eight behavioral test vectors. If the target hardware uses *carry look-ahead* circuits, additional behavioral faults are defined.

5.2 Subtraction

Subtraction is closely related to addition and all the techniques previously discussed in this chapter are applicable. Also, the subtraction operation is often implemented indirectly using adders and 2's complement arithmetic. A *ripple borrow* subtractor and a 2's complement addition will be used to demonstrate extension of functional testing to the subtraction operation.

5.2.1 Direct Subtraction

Subtraction, like addition, can be performed by the interconnection of functional modules. Consider the subtraction operation $D \leq M - S$. The *difference* (D) is formed by the subtraction of the *subtrahend* (S) from the *minuend* (M). Like *ripple carries* in addition, *full subtractors* (FS) and *half subtractors* (HS) can be interconnected via *borrows* between stages.

The truth table for a *full subtractor* is presented in Table 5-15, where BI and BO represent *borrow-in* and *borrow-out* respectively [35]. Like the *half adder*, the *half subtractor* represents a subset of the *full subtractor*, where $BI = 0$.

Test #	BI	M	S	D	BO
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	0
4	1	0	0	1	1
5	1	0	1	0	1
6	1	1	0	0	0
7	1	1	1	1	1

Table 5-15 Truth table for full subtractor.

5.2.1.1 Functional Testing

The functional testing strategy presented for the *ripple carry* adder can now be applied to direct subtraction. Complete functional testing can be achieved by exhaustive testing of each subtractor module.

The functional test vectors proposed for direct subtraction are presented in Table 5-16. The Phase I tests represent complete testing of the *half subtractor* (HS_0) module. Phase II continues testing of the *full subtractor* (FS_i) modules where $BI = 1$. Finally, Phase III completes testing of the *full subtractor* modules by interleaving the remaining tests from Table 5-15.

Phase	M	S	D	Test #			
				FS_3	FS_2	FS_1	HS_0
I	0000	0000	0000	0	0	0	0
	0000	1111	0001	5	5	5	1
	1111	0000	1111	2	2	2	2
	1111	1111	0000	3	3	3	3
II	0000	0001	1111	4	4	4	1
	1110	1111	1111	7	7	7	1
III	0101	1010	1011	1	6	1	2
	1010	0101	0101	6	1	6	1

Table 5-16 Functional tests for 4-bit direct subtraction.

5.2.1.2 Application of the Behavioral Test Vectors

Again, the behavioral fault model follows directly from the error vector approach presented earlier in this chapter. Behavioral test vectors are derived from the error vectors formed by the functional tests from Table 5-16.

A 4-bit subtractor will now be used to evaluate the effectiveness of the behavioral test vectors. The VHDL behavioral description for example SUB4 is shown in Figure 5-13. The example subtracts B from A to produce a 4-bit *difference D*.

```

architecture behave of sub4 is
begin
  process(A,B)
  begin
    D <= A - B;
  end process;
end behave;
```

Figure 5-13 Behavioral description for example SUB4.

Example SUB4 was synthesized with AutoLogic II to produce the subtraction circuit shown in Figure 5-14. Fault simulations were then performed using the behavioral test vectors derived from Table 5-16. As expected, the behavioral test vectors achieved a *SSL* gate level fault coverage of $112/112 = 100\%$.

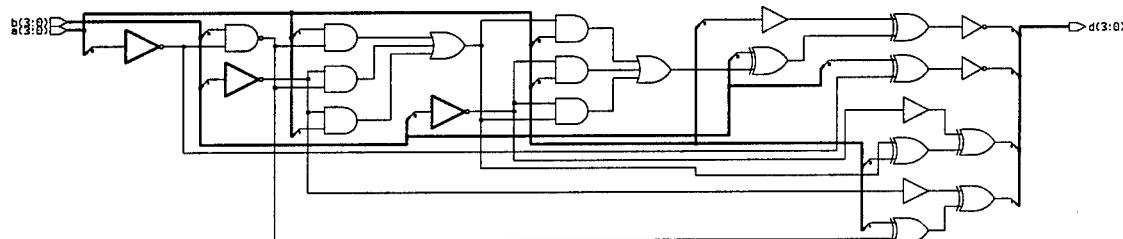


Figure 5-14 Synthesized circuit for example SUB4.

5.2.2 Subtraction Using Addition Circuitry

Subtraction can also be performed by taking the negative of the *subtrahend* and performing an addition [35][41]. With 2's complement arithmetic, the negation can be performed by taking the logical complement of *B* and adding 1. Thus, the subtraction operation becomes:

$$A - B = A + (-B) = A + \bar{B} + 1$$

The behavioral test vectors developed for direct subtraction can now be evaluated for their performance on a subtractor realized with addition circuitry. A block diagram for a 4-bit subtractor implemented with *full adders* is presented in Figure 5-15.

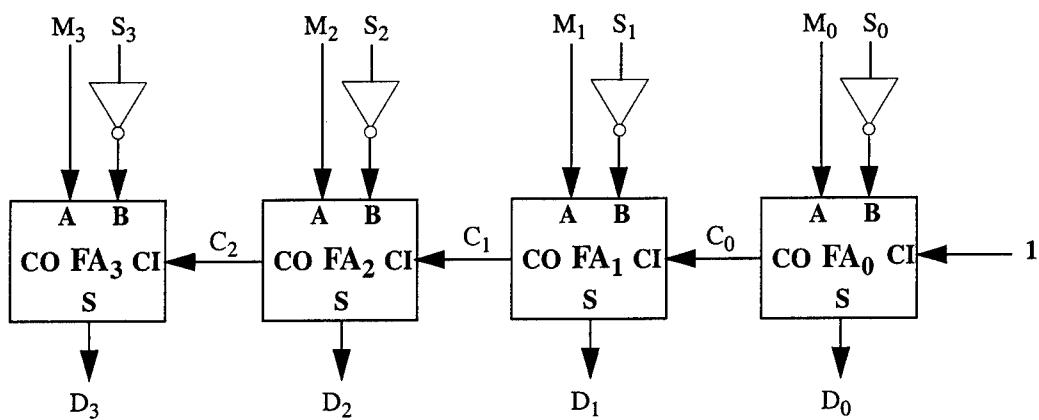


Figure 5-15 Subtractor implemented with full adders.

Table 5-17 shows the functional tests applied to the *full adders* (FA_i) by the subtraction test vectors from Table 5-16. Since $CIN = 1$ for all vectors, **Test 4** through **Test 7** represents all possible tests for FA_0 . Hence, complete functional testing for each stage of the adder is achieved by the behavioral test vectors for subtraction. Additional fault simulation results can be found in Appendix B.

Phase	Subtraction			Addition				Test #			
	M	S	D	A	B	CIN	S	FA ₃	FA ₂	FA ₁	FA ₀
I	0000	0000	0000	0000	1111	1	0000	5	5	5	5
	0000	1111	0001	0000	0000	1	0001	0	0	0	4
	1111	0000	1111	1111	1111	1	1111	7	7	7	7
	1111	1111	0000	1111	0000	1	0000	6	6	6	6
II	0000	0001	1111	0000	1110	1	1111	1	1	1	4
	1110	1111	1111	1110	0000	1	1111	2	2	2	4
III	0101	1010	1011	0101	0101	1	1011	4	3	4	7
	1010	0101	0101	1010	1010	1	0101	3	4	3	4

Table 5-17 Functional tests for adder by subtraction test vectors.

5.2.3 Summary

A behavioral fault model for subtraction has been derived using a complete functional testing technique. Though based on direct subtraction, the behavioral test vectors also provide complete gate level fault coverage when implemented with adders. The functional tests can be readily extended to n -bits, still requiring only eight behavioral test vectors. Supplemental behavioral faults for a *CLA* implementation can be easily derived via the relationships of 2's complement arithmetic. Such an example can be found in Appendix A.

5.3 Constants as Operands

When one *operand* for an *arithmetic operator* is a *constant*, the behavior changes to that of a *unary operator* such as *increment* or *decrement*. Controllability is lost over the constant's input patterns, so the previously developed functional tests cannot be applied.

A 4-bit increment function, $Z \leq Y + 1$, is used here as an example. Applying the constant *0001* to a *ripple carry* adder structure, the FA equations can be rewritten as:

$$\begin{array}{ll} S_0 = \bar{A}_0 & S_i = A_i \oplus C_{i-1} \\ C_0 = A_0 & C_i = A_i C_{i-1} \end{array}$$

Stage 0 (ST_0) of the resulting incrementor is a single input function, while the remaining stages are 2-input functions that can be recognized as *half adders*. A functional testing strategy, like that in Section 5.1.1.1, can now be applied.

5.3.1 Functional Testing

Complete functional testing of the *unary operator increment* will now be achieved by exhaustive testing of every stage. The test vectors and resulting tests for each stage are presented in Table 5-18.

Phase	Y	Z	Test #			
			HA ₃	HA ₂	HA ₁	ST ₀
I	0000	0001	0	0	0	0
	1111	0000	3	3	3	1
II	1110	1111	2	2	2	0
	1101	1110	2	2	1	1
	1011	1100	2	1	3	1
	0111	1000	1	3	3	1

Table 5-18 Functional tests for 4-bit increment function.

As can be seen from the patterns in the test results, complete functional testing of an n -bit *increment* function can be achieved with $n+2$ test vectors. Similar analysis on a 4-bit *decrement* function, $Z \leq Y - 1$, achieves similar results.

5.3.2 Generalized Behavioral Fault Model

By examining the functional test vector patterns for each *unary operator* function, $(Y + 1, Y + 2, \dots)$ and $(Y - 1, Y - 2, \dots)$, a generalized behavioral fault model can be developed. In the case of positive increments, the n test vectors for the Phase II tests can be derived by starting with a test pattern formed by the complement of the increment value.

For example, a 5-bit implementation of $Z \leq Y + 3$ would start with a Phase II test vector of 11100 . The starting test pattern is then rotated $n-1$ times to produce the remaining behavioral test vectors.

For negative numbers, the starting test pattern is based on the positive representation of the number. For example, a 6-bit implementation of $Z \leq Y - 5$ would start with a Phase II test pattern of 000101 . The functional tests produced by the resulting behavioral test vectors are summarized in Table 5-19.

Phase	Y	Z	Test #					
			ST ₅	ST ₄	ST ₃	ST ₂	ST ₁	ST ₀
I	000000	111011	0	0	0	0	0	0
	111111	111010	3	3	3	3	3	1
II	000101	000000	1	1	1	3	1	1
	001010	000101	1	1	2	1	2	0
	010100	001111	1	2	0	2	0	0
	101000	100011	3	1	2	0	0	0
	010001	001100	1	2	0	1	1	1
	100010	011101	2	0	0	1	2	1

Table 5-19 Functional tests for 6-bit function $Z \leq Y - 5$.

Complete functional testing, of the *unary operators* formed by *arithmetic operators* with a *constant operand*, can be achieved with $n+2$ behavioral test vectors. The resulting gate level fault coverage, however, depends on optimizations performed by a synthesis tool which may affect the underlying *ripple carry* structure. Further application of these new behavioral fault models can be found in the examples in Appendix A.

5.4 Comparison with Previous Fault Models

As was the case with *relational operators*, few previous fault models address *arithmetic operators*. As part of the development of the “B-algorithm: A Behavioral Test Generation Algorithm,” [21][22] Cho and Armstrong developed a new behavioral fault model which included *arithmetic operators* as part of *micro-operation* faults. Such faults perturb an *arithmetic operator* to another operator as indicated in Table 5-20.

Fault-free Operator	Faulty Operator
ADD	SUB, XOR
SUB	ADD, XOR

Table 5-20 Micro-operation Faults

Recall that this fault model suffered from what the authors called the “big micro-operation problem,” where only a single test vector is generated for a complex block of logic. To raise their equivalent gate level fault coverage to acceptable levels, an additional $4n-1$ test vectors are generated by a heuristic test generator for each n -bit micro-operation.

A similar fault model was proposed by Al Hayek and Robach [4] as part of a mutation-based testing strategy in which VHDL behavioral faults are considered as software faults. For Arithmetic Operator Replacement (*AOR*), *ADD* (+) is replaced by *SUB* (-). Mutation analysis does not take into account the size of the hardware implementation, because it considers each *arithmetic operator* as a software operation and consequently generates only one test vector. In order to improve the performance of their technique, the authors also resorted to heuristics to generate additional test vectors for their complex operators.

The new behavioral fault models developed for *arithmetic operators* eliminate the need to supplement test vector sets. While the new fault models are definitely more complex than previous ones, this is because they more accurately reflect the underlying complexity of the hardware faults which they attempt to model.

5.5 Conclusions

New behavioral fault models have been developed for the predefined VHDL *arithmetic operators* *ADD* (+) and *SUB* (-). The fault models are based on complete functional testing of arithmetic building blocks. Though optimizations may modify the building blocks, the basic function of the *arithmetic operators* remains unchanged. By concentrating on functional testing, complete gate level *SSL* fault coverage can be obtained over a broad range of hardware implementations.

The base fault model is derived from the *ripple carry* connection of *full* and *half adders*. Because such realizations are made up of 2- and 3-input functional building blocks, complete functional testing can be achieved by exhaustive testing of each module.

Only eight behavioral test vectors are required for complete gate level fault coverage, regardless of the size of the *operands*.

If the target hardware uses *carry look-ahead* circuits, additional behavioral faults are defined. Testing for all the faults in the highest order carry in a *CLA* adder will provide coverage for all lower order faults. Test vectors are added for *missing carry* faults and *extra carry* faults for specific stages. Knowledge of the target architecture will allow optimization of the test vector set while still achieving complete gate level fault coverage.

Subtraction is closely related to addition and all the techniques previously discussed are applicable. The functional tests can be readily extended to n -bits, still requiring only eight behavioral test vectors. Though based on direct subtraction, they also provide complete functional testing when the operation is realized indirectly with adders. Additional behavioral faults for a *CLA* implementation can be easily derived via the relationships of 2's complement arithmetic.

The new behavioral fault models developed for *arithmetic operators* eliminate the need to supplement test vector sets via methods such as heuristics. While the new fault models are more complex than previous ones, this is because they more accurately reflect the underlying complexity of the hardware faults which they attempt to model.

Chapter 6

Other Operators

The remaining VHDL *operators* from Table 1-2 include *logical*, *unary*, *multiplying*, and *miscellaneous*. *Logical operators* provide a close link between behavioral and gate level descriptions. Mapping *SSL* gate level faults into the behavioral domain is, therefore, a fairly straight forward process. However, differences in actual gate level structures, due to optimization and synthesis tools, must also be taken into account.

In contrast to previously discussed *binary operators*, *unary operators* affect only a single *operand*. This distinction does not alter the analysis of gate level faults and their mapping to behavioral faults. Though classified in the *miscellaneous* category, the *operator ABS* will be considered with this group.

Due to power of 2 restrictions placed on *multiplying operators*, detailed in Figure 1-1, implementation becomes simply a shifting of lines, rather than any additional hardware. Since no more gates are implied by such *operations*, no additional gate level faults are introduced. The same synthesis guidelines apply to the miscellaneous operator (**), hence, no behavioral faults will be defined.

6.1 Logical Operators

The predefined VHDL *logical operators* include *AND*, *OR*, *NAND*, *NOR*, and *XOR*. All these *operators* are binary, therefore only 2-input gate level structures are implied. The *miscellaneous operator NOT* is a *unary operator* that does not introduce additional *SSL* gate level faults, not covered by other *operators*. Further, the logical pairs *AND/OR* and *NAND/NOR* differ by only a single inversion, hence detailed analysis of one group will provide the insight necessary to develop the behavioral fault models for the entire set of *operators*.

The remaining *logical operator*, *XOR*, will be examined separately due to the unique nature of its functional faults. Additionally, previous research on *XOR* structures, such as parity trees, can provide optimizations for the behavioral tests necessary to provide complete gate level fault coverage.

6.1.1 AND/OR

The *logical operators AND/OR* provide a close link between behavioral and gate level descriptions. Mapping *SSL* gate level faults into the behavioral domain is, therefore, a fairly straight forward process. However, differences in actual gate level structures, due to optimization and synthesis tools, must also be taken into account.

6.1.1.1 Functional Faults

The behavioral description of a 2-*operand AND operation* can be expressed as:

$Z \leq A \text{ AND } B;$

A direct gate level implementation results in a 2-input *AND* gate which can be analyzed for *SSL* faults. A reduced set of functional faults for the *AND operation* is presented in Table 6-1. The three test vectors (*AB*) necessary to detect all functional faults are, therefore, *01*, *10*, and *11*.

A	B	Z	$Z = 0$	$Z = A$	$Z = B$
0	0	0			
0	1	0			1
1	0	0		1	
1	1	1	0		

Table 6-1 Functional faults for AND operation.

A similar analysis of the *OR operation* produces the reduced set of functional faults shown in Table 6-2. The required functional test vectors are *00*, *01*, and *10*.

A	B	Z	$Z = 1$	$Z = A$	$Z = B$
0	0	0	1		
0	1	1		0	
1	0	1			0
1	1	1			

Table 6-2 Functional faults for OR operation.

The functional faults for the *NAND* and *NOR* operators follow directly from the above analysis. The functional tests for the *NAND* operation are the same as for the *AND* operation, while the tests for the *NOR* operation are the same as the *OR*.

6.1.1.2 Complex Expressions

The functional faults for a single *logical operator* have been determined quite easily. However, interactions among these *operators* in more complex *expressions* must also be addressed. Since all the *logical operators* are binary, these interactions can be investigated with the use of a binary tree.

Consider the behavioral description of a logical *expression* presented in Figure 6-1.

```

entity SOP1 is
    port( A, B, C, D: in std_logic;
          Z: out std_logic );
end SOP1;

architecture behave of SOP1 is
begin
    process(A,B,C,D)
    begin
        Z <= (A AND B) OR (C AND D);
    end process;
end behave;
```

Figure 6-1 Behavioral description for example SOP1.

The *expression* on the right hand side of the assignment statement can be parsed into a binary tree shown in Figure 6-2. The nodes (1,2,3) are formed by the *logical operators*, while the leaves of the tree are the signals *A*, *B*, *C*, and *D*.

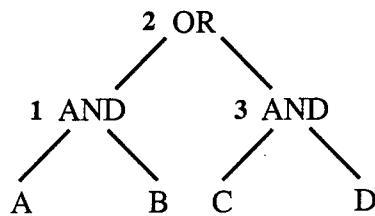


Figure 6-2 Binary tree representing example SOP1.

According to the previous analysis, three functional faults affect each of the three *logical operators* in the *expression*. For example, the first functional fault to the *AND* opera-

tor at node 1 requires a test vector (AB) of 01 and produces an erroneous output of $A \text{ AND } B = 1$. In order for this erroneous output to be observable at the *expression* output Z , appropriate values for the other signals (C,D) must be determined. This can be accomplished using the following set of Boolean identities:

$$\begin{aligned} Y \text{ OR } 0 &= Y \\ Y \text{ OR } 1 &= 1 \end{aligned}$$

$$\begin{aligned} Y \text{ AND } 1 &= Y \\ Y \text{ AND } 0 &= 0 \end{aligned}$$

According to the first identity, setting the right hand *operand* of the *OR operator* at node 2 to 0 will allow the left hand *operand* to propagate up the tree unchanged. Using the last identity, setting either *operand* of the *AND operator*, at node 3, to 0 will produce the desired input to node 2. Applying this set of identities along with the functional faults for the *AND* and *OR operators* produces the test vectors shown in Table 6-3.

Node	Functional Test	Test Requirements	Test Vector (ABCD)
1	01	$A = 0, B = 1, (C \text{ AND } D) = 0$	010X
	10	$A = 1, B = 0, (C \text{ AND } D) = 0$	100X
	11	$A = 1, B = 1, (C \text{ AND } D) = 0$	110X
2	00	$(A \text{ AND } B) = 0, (C \text{ AND } D) = 0$	0X0X
	01	$(A \text{ AND } B) = 0, (C \text{ AND } D) = 1$	0X11
	10	$(A \text{ AND } B) = 1, (C \text{ AND } D) = 0$	110X
3	01	$C = 0, D = 1, (A \text{ AND } B) = 0$	0X01
	10	$C = 1, D = 0, (A \text{ AND } B) = 0$	0X10
	11	$C = 1, D = 1, (A \text{ AND } B) = 0$	0X11

Table 6-3 Functional test vectors for example SOP1.

The set of functional test vectors can be reduced by combination of the don't care values. The final test vectors for example SOP1 are presented in Table 6-4. Also listed are the functional tests covered for each node of the parse tree.

Test Vector (ABCD)	Functional Tests (Node [Test])
0101	1[01], 2[00], 3[01]
0X10	3[10]
0X11	2[01], 3[11]
100X	1[10]
110X	1[11], 2[10]

Table 6-4 Reduced test vectors for example SOP1.

6.1.1.3 Scalability

Example SOP4 was created by expanding the width of the signals A , B , C , and D to `std_logic_vector(3 downto 0)`. It has been previously shown that expansion of the data path simply causes replication of the single bit case implemented in parallel. Hence, extra test vectors are not required since the additional hardware can be tested at the same time. The expanded test vectors for example SOP4 are shown in WAVES format in Figure 6-3.

```
% A   B   C   D   Z   : time ;
0000 1111 0000 1111 0000 : 500 ns;
0000 XXXX 1111 0000 0000 : 500 ns;
0000 XXXX 1111 1111 1111 : 500 ns;
1111 0000 0000 XXXX 0000 : 500 ns;
1111 1111 0000 XXXX 1111 : 500 ns;
```

Figure 6-3 WAVES test vectors for example SOP4.

6.1.1.4 Behavioral Fault Model

As was the case with the *arithmetic operators*, an error vector approach is taken for abstracting the *logical operator* functional faults into the behavioral domain. The *XOR operator* provides the desired corruption properties by inverting the appropriate bit(s) when presented with a non-zero error vector.

Recall from Table 6-3, the first functional fault to the *AND operator* at node 1 required that $A = 0$, $B = 1$, and $(C \text{ AND } D) = 0$. This fault criteria can be directly translated to an error vector for example SOP1 as:

```
Z <= (A AND B) OR (C AND D) XOR (A = '0' AND B = '1'
AND (C AND D) = '0')
```

Again, as was the case for the *arithmetic operators*, this expression is not syntactically correct due to VHDL type differences. A complete implementation of the behavioral fault model for example SOP1 is presented in Figure 6-4.

```

Z <= (A AND B) OR (C AND D) XOR to_stdlogic(
  (A = '0' AND B = '1' AND (C AND D) = '0') OR
  (A = '1' AND B = '0' AND (C AND D) = '0') OR
  (A = '1' AND B = '1' AND (C AND D) = '0') OR
  ((A AND B) = '0' AND (C AND D) = '0') OR
  ((A AND B) = '0' AND (C AND D) = '1') OR
  ((A AND B) = '1' AND (C AND D) = '0') OR
  (C = '0' AND D = '1' AND (A AND B) = '0') OR
  (C = '1' AND D = '0' AND (A AND B) = '0') OR
  (C = '1' AND D = '1' AND (A AND B) = '0'));
```

Figure 6-4 Behavioral fault model for example SOP1.

6.1.1.5 Application of the New Fault Models

A simple example is now presented to demonstrate the application of the new fault models developed for *logical operators*. A test vector set will be formed based on behavioral faults and then applied to synthesized gate level realizations.

Example GT in Figure 6-5 uses *logical operators* to describe a Boolean *expression* for the 2-bit *greater than* function, examined in detail in Chapter 4. Example GT presents two minor differences from example SOP1. First, the inclusion of the *unary operator NOT* means that the resulting parse tree will not be completely binary. This should have no

```

entity gt is
  port(A, B: in std_logic_vector(1 downto 0);
       GT: out std_logic );
end gt;

architecture behave of gt is
begin
  process(A,B)
  begin
    GT <= (A(1) AND not B(1)) OR (A(0) AND not B(0)
                                    AND (A(1) OR not B(1)));
  end process;
end behave;
```

Figure 6-5 Behavioral description for example GT.

effect on the test generation process since functional faults are only defined for the binary *logical operators*. Second, not all of the pairings are explicitly defined for the binary *logical operators* in example GT. Hence, the precise implementation by the synthesis tool cannot be determined. Since the new fault models are based on a functional analysis, they should provide complete gate level fault coverage over a broad range of realizations.

A parse tree for example GT is presented in Figure 6-6. As before, the binary nodes are formed by the *logical operators*, while the leaves of the tree represent the signals.

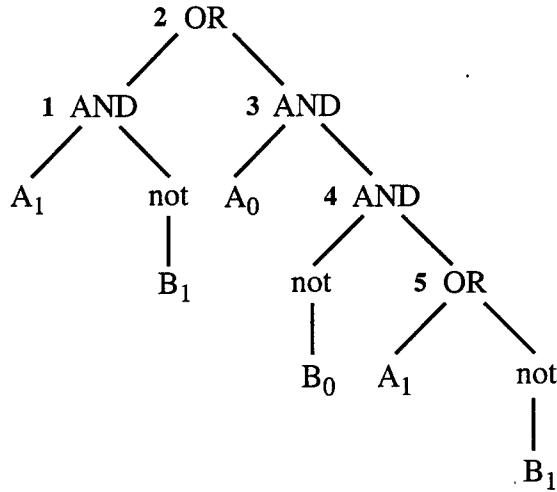


Figure 6-6 Parse tree for example GT.

Application of the new behavioral fault models implies three behavioral faults for each of the five binary *logical operators*. The resulting test vectors are presented in Table 6-5. Again, the set of behavioral test vectors can be reduced by combination of the don't care values. The final test vectors for example GT are presented in Table 6-6. It is worth noting that the behavioral test vectors derived here are consistent with the functional faults for the 2-bit GT function presented in Figure 4-2.

Example GT was synthesized with AutoLogic II to produce the gate level Structure1 shown in Figure 6-7. Note that the groupings for the *AND* gates do not match the parse tree in Figure 6-6. According to MIL-STD 883D, Structure1 contains 30 unique gate level SSL faults. Fault simulations using the behavioral test vectors from Table 6-6 resulted in complete gate level fault coverage.

Node	Behavioral Test	Test Requirement	Test Vector (A ₁ A ₀ B ₁ B ₀)
1	01	A ₁ = 0, not B ₁ = 1, A ₀ AND not B ₀ AND (A ₁ OR not B ₁) = 0	000X
	10	A ₁ = 1, not B ₁ = 0, A ₀ AND not B ₀ AND (A ₁ OR not B ₁) = 0	101X
	11	A ₁ = 1, not B ₁ = 1, A ₀ AND not B ₀ AND (A ₁ OR not B ₁) = 0	100X
2	00	A ₁ AND not B ₁ = 0, A ₀ AND not B ₀ AND (A ₁ OR not B ₁) = 0	00XX
	01	A ₁ AND not B ₁ = 0, A ₀ AND not B ₀ AND (A ₁ OR not B ₁) = 1	0100
	10	A ₁ AND not B ₁ = 1, A ₀ AND not B ₀ AND (A ₁ OR not B ₁) = 0	100X
3	01	A ₀ = 0, not B ₀ AND (A ₁ OR not B ₁) = 1, A ₁ AND not B ₁ = 0	1010
	10	A ₀ = 1, not B ₀ AND (A ₁ OR not B ₁) = 0, A ₁ AND not B ₁ = 0	01X1
	11	A ₀ = 1, not B ₀ AND (A ₁ OR not B ₁) = 1, A ₁ AND not B ₁ = 0	1110
4	01	not B ₀ = 0, A ₁ OR not B ₁ = 1, A ₀ = 1, A ₁ AND not B ₁ = 0	1111
	10	not B ₀ = 1, A ₁ OR not B ₁ = 0, A ₀ = 1, A ₁ AND not B ₁ = 0	0110
	11	not B ₀ = 1, A ₁ OR not B ₁ = 1, A ₀ = 1, A ₁ AND not B ₁ = 0	1110
5	00	A ₁ = 0, not B ₁ = 0, not B ₀ = 1, A ₀ = 1, A ₁ AND not B ₁ = 0	0110
	01	A ₁ = 0, not B ₁ = 1, not B ₀ = 1, A ₀ = 1, A ₁ AND not B ₁ = 0	0100
	10	A ₁ = 1, not B ₁ = 0, not B ₀ = 1, A ₀ = 1, A ₁ AND not B ₁ = 0	1110

Table 6-5 Behavioral test vectors for example GT.

Test Vector (A ₁ A ₀ B ₁ B ₀)	Functional Tests (Node [Test])
000X	1[01], 2[00]
0100	2[01], 5[01]
01X1	3[10]
0110	4[10], 5[00]
100X	1[11], 2[10]
1010	1[10], 3[01]
1110	3[11], 4[11], 5[10]
1111	4[01]

Table 6-6 Reduced test vectors for example GT.

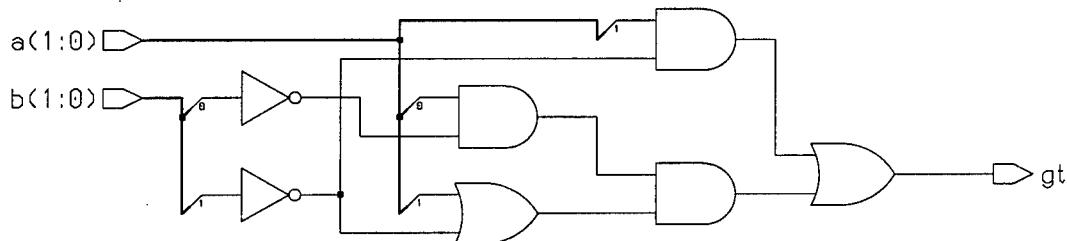


Figure 6-7 Synthesized Structure1 for example GT.

Example GT was next synthesized and optimized for an alternate target technology. The resulting Structure2 is shown in Figure 6-8. Fault simulations using the same behavioral test vectors achieved a SSL gate level fault coverage of $35/35 = 100\%$. The fault coverage graph is shown in Figure 6-9.

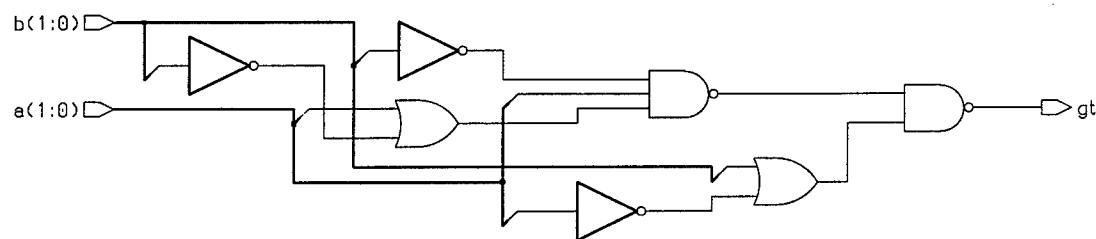


Figure 6-8 Synthesized Structure2 for example GT.

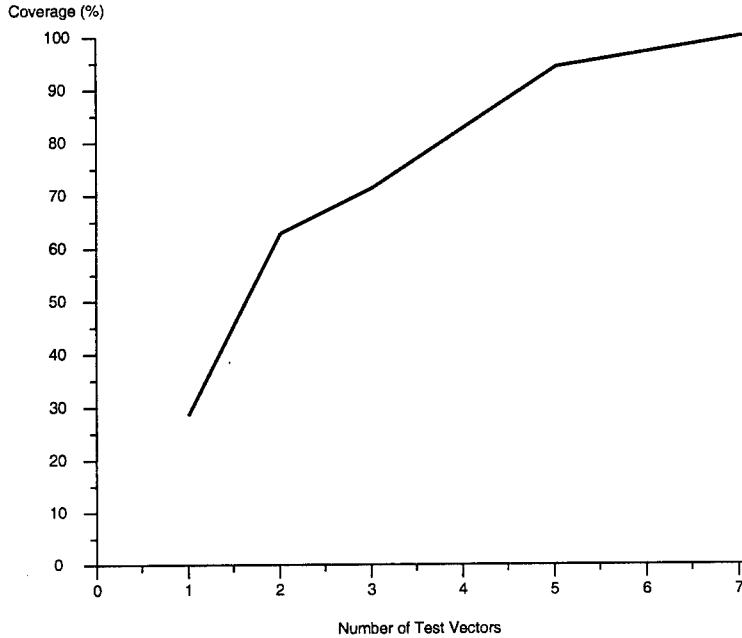


Figure 6-9 Fault coverage for Structure2 of example GT.

6.1.2 XOR

The remaining *logical operator*, *XOR*, is examined here separately, due to the unique nature of its functional faults. Additionally, previous research on *XOR* structures, such as parity trees [16][53], can provide optimizations for the behavioral tests necessary to provide complete gate level fault coverage.

6.1.2.1 Functional Faults

The *XOR* gate has numerous logical implementations producing several different sets of functional faults. In order to test an *XOR* gate whose internal structure is unknown, an exhaustive test set, four patterns, is needed to detect all *SSL* faults [30]. A generalized set of functional faults is presented in Table 6-7.

The set of generalized functional faults and the following Boolean identities allow functional tests to be generated for the *XOR operator* used in complex *expressions* with the other *logical operators*.

$$\begin{aligned} Y \text{ XOR } 0 &= Y \\ Y \text{ XOR } Y &= 0 \\ Y \text{ XOR } \bar{Y} &= 1 \end{aligned}$$

Though not optimal, the functional tests ensure complete *SSL* fault coverage of the *XOR* gates regardless of their internal structure.

	A	B	Z	$Z = A \text{ NAND } B$	$Z = A \text{ AND } \bar{B}$	$Z = \bar{A} \text{ AND } B$	$Z = A \text{ OR } B$
0	0	0	1				
0	1	1			0		
1	0	1				0	
1	1	0					1

Table 6-7 Generalized functional faults for XOR operation.

6.1.2.2 Optimized Test Generation

Expressions containing only *XOR operators* are common for circuits such as parity networks. Previous research has demonstrated the generation of optimum test patterns for such parity networks with fixed structures [16][53]. However, when working with behavioral descriptions, the actual gate level structure is often unknown and may ultimately be determined by a synthesis tool. Still, modifications to the algorithms are possible to allow for complete gate level fault coverage over a broad range of implementations.

The Bossen algorithm [16] applies an exhaustive test to each *XOR* gate by using the labeling scheme shown in Figure 6-10. The test sequences are labeled R, S, and T. Each of the sequences is the modulo-2 sum of the other two. That is, $T = R \oplus S$, $S = T \oplus R$, and $R = S \oplus T$.



Figure 6-10 Labeling scheme for Bossen test.

Test generation using the Bossen algorithm will be demonstrated via a simple example. The behavioral description for XOR5 is shown below:

```
Z <= A XOR B XOR C XOR D XOR E
```

Grouping the terms from left to right produces a linear tree or cascade implementation shown in Figure 6-11.

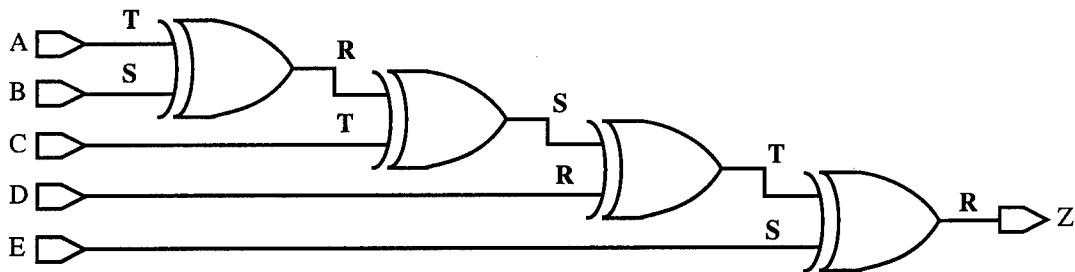


Figure 6-11 Structure Cascade1 for example XOR5.

The test set is developed by first assigning one of the three sequences to the output of the tree. Other sequences are then determined moving from right to left, ensuring that each *XOR* gate is exhaustively tested. The resulting test vectors are shown in Table 6-8. Note that the labeling sequences are not unique and the test vectors generated may not completely test other implementations of the same expression.

Signal	Label	Sequence
A	T	0101
B	S	0011
C	T	0101
D	R	0110
E	S	0011

Test Vector (ABCDE)	Z
00000	0
10110	1
01011	1
11101	0

Table 6-8 Bossen test vectors for Cascade1.

If the behavioral description for XOR5 is instead implemented by grouping terms from right to left, a second cascade structure is formed. Figure 6-12 shows that the labeling scheme from Figure 6-11 cannot be applied to this alternate implementation.

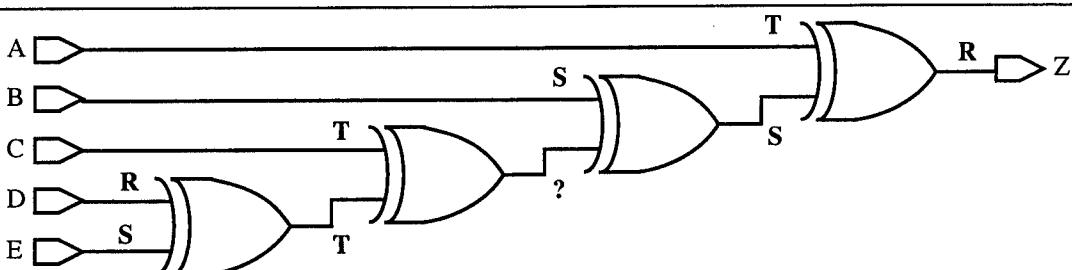


Figure 6-12 Structure Cascade2 for example XOR5.

In fact, no labeling will provide exhaustive testing for all *XOR* gates in both circuits Cascade1 and Cascade2. What is needed is a modification to the Bossen algorithm that will take into account the most likely implementations of an *XOR*-only *expression*. Consider the following groupings for example XOR5:

$$Z \leq (((A \text{ XOR } B) \text{ XOR } C) \text{ XOR } D) \text{ XOR } E \quad (\text{Cascade1})$$

$$Z \leq ((A \text{ XOR } B) \text{ XOR } (C \text{ XOR } D)) \text{ XOR } E \quad (\text{Balanced1})$$

$$Z \leq A \text{ XOR } (B \text{ XOR } (C \text{ XOR } (D \text{ XOR } E))) \quad (\text{Cascade2})$$

$$Z \leq A \text{ XOR } ((B \text{ XOR } C) \text{ XOR } (D \text{ XOR } E)) \quad (\text{Balanced2})$$

The first and third groupings produce structures Cascade1 and Cascade2 respectively, while the second and forth groupings produce balanced trees [53]. If we consider these structures to represent a broad range of possible implementations of the behavioral description, a generalized Bossen algorithm can then provide complete gate level fault coverage.

An extra restriction is added to the Bossen algorithm to account for multiple possible structures: According to the assumed groupings for a cascade and its corresponding balanced implementation (Cascade *i*, Balanced *i*), no two inputs that are grouped together can be assigned the same test sequence.

Applying this modified algorithm to structure Cascade2 produces the labeling sequences shown in Figure 6-13 and the test vectors shown in Table 6-9. Since the test vectors developed for structure Cascade1 already meet the additional restriction for the modified Bossen algorithm, the generalized set of test vectors from Table 6-8 and Table 6-9 will now provide exhaustive testing for all four structures of XOR5.

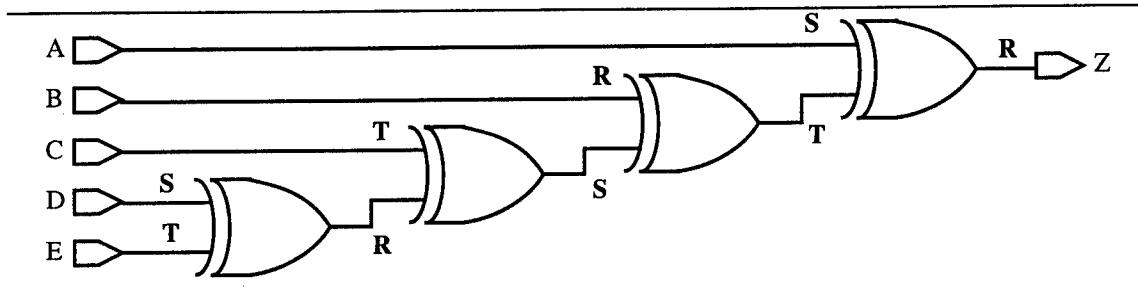


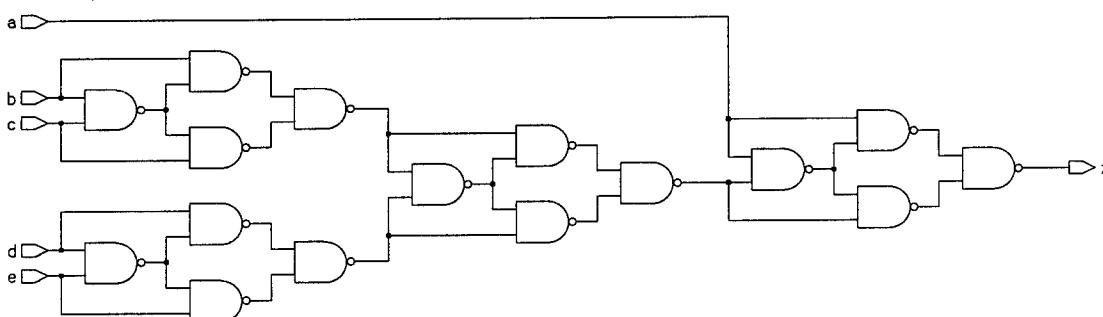
Figure 6-13 Modified Bossen test for Cascade2.

Signal	Label	Sequence	Test Vector (ABCDE)	Z
A	S	0011	00000	0
B	R	0110	01101	1
C	T	0101	11010	1
D	S	0011	10111	0
E	T	0101		

Table 6-9 Modified Bossen test vectors for Cascade2.**6.1.2.3 Evaluation of the Generalized Test Vectors**

The generalized Bossen test vectors for example XOR5 are presented in WAVES format in Figure 6-14. Fault simulations were conducted on multiple implementations including Structure4 in Figure 6-15. Complete SSL gate level fault coverage was achieved for each realization. An example fault coverage graph for Structure4 is shown in Figure 6-16. Note that though the behavioral test vectors are optimized compared to those generated by a parse tree, they are still generalized to apply to multiple gate level structures.

```
% ABCDE Z : time ;
00000 0 : 500 ns;
01011 1 : 500 ns;
01101 1 : 500 ns;
10110 1 : 500 ns;
10111 0 : 500 ns;
11010 1 : 500 ns;
11101 0 : 500 ns;
```

Figure 6-14 Generalized Bossen test vectors for example XOR5.**Figure 6-15 Structure4 for example XOR5.**

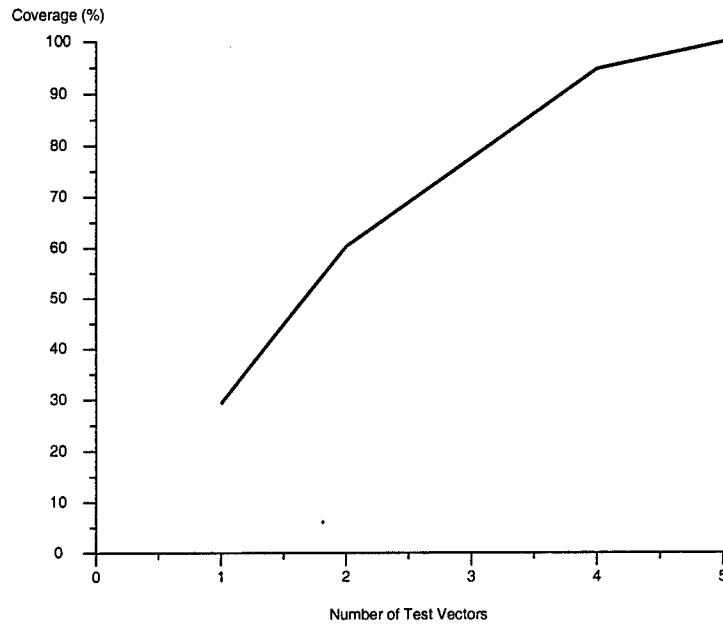


Figure 6-16 Fault coverage for Structure4 of example XOR5.

6.1.3 Comparison with Previous Fault Models

Behavioral fault models for *logical operators* are addressed by Armstrong et al., where early models replaced one micro-operation with any other in its class [9][69], while later studies tried to determine which perturbations produced the best fault coverage [19]. Finally, the B-algorithm eliminated micro-operation faults for *logical operators* by defining bit-wise *stuck-at* faults for any one of its arguments (a *signal* or an unnamed *signal* for an *expression*) [21][22]. This method amounts to exhaustive testing of each *logical operator* in an *expression*.

In their mutation based testing strategy, Al Hayek and Robach [4] define Logical Operator Replacement (*LOR*) in which each *logical operator* is replaced by each of the other operators. This method treats the VHDL description as software and has little relation to actual hardware faults. Finally, other fault models [18][27][60] completely neglect *logical operators* and instead rely on *stuck-signals* to provide fault coverage.

The new behavioral fault models developed for *logical operators* are based on functional faults that require less than exhaustive testing for all operators except *XOR*. For the special case of *XOR*-only *expressions*, a generalized Bossen algorithm is presented that allows for optimization of test sequences. The new fault models and algorithms thus provide complete *SSL* gate level fault coverage for a broad range of implementations.

6.2 Unary Operators

The *unary operator* for negation (-) performs a function related to the *miscellaneous operator ABS*; each affects the sign of a 2's complement number. By analyzing each *operator*'s functional faults in tandem with the other, a consistent behavioral fault model can be developed. Note that no previous behavioral fault models even address these *operators*.

6.2.1 Absolute Value

The *ABS operator* computes the absolute value of a 2's complement number. The operator's functional faults will be investigated for the 4-bit case, then generalized to n-bits. The following Boolean equations describe the 4-bit absolute value function:

$$\begin{aligned} Z_0 &= X_0 \\ Z_1 &= X_1 \oplus (X_3 X_0) \\ Z_2 &= X_2 \oplus (X_3 X_1 + X_3 X_0) \\ Z_3 &= 0 \end{aligned}$$

Analysis of the faulty behavior of $Z = ABS X$ produces the reduced set of functional faults shown in Table 6-10. The faults are generically labeled F1-F5 and are shown with the integer value of the appropriate test vectors. Before proceeding further with the development of a behavioral fault model, the *negation operator* will first be examined for common functional faults.

Functional Fault	Test Vectors (integer)
F1	1, 3, 5, 7
F2	2, 3, 6, 7
F3	-3, -7
F4	-2, -6
F5	-4

Table 6-10 Reduced functional faults for 4-bit ABS.

6.2.2 Negation

The *negation operator* (-) changes the sign of a 2's complement number. The following Boolean equations describe the 4-bit negation function:

$$\begin{aligned}
 Z_0 &= X_0 \\
 Z_1 &= X_1 \oplus X_0 \\
 Z_2 &= X_2 \oplus (X_1 + X_0) \\
 Z_3 &= X_3 \oplus (X_2 + X_1 + X_0)
 \end{aligned}$$

Analysis of the faulty behavior of $Z = -X$ produces the reduced set of functional faults shown in Table 6-11. The faults are generically labeled F6-F10 and are shown with the integer value of the appropriate test vectors.

Functional Fault	Test Vectors (integer)
F6	0
F7	-3, -7, 1, 5
F8	-5, -6, -7, 1, 2, 3
F9	-2, -6, 2, 6
F10	-4, 4

Table 6-11 Reduced functional faults for 4-bit negation.

6.2.3 Generalized Functional Faults

Examination of the reduced set of functional faults for the *absolute value* and *negation operators* provides the necessary insight for developing a common fault model. A generalized set of functional faults is shown in Table 6-12. The faults are covered by three tests spanning the range of integer values (0, -7, 7) combined with a readily identifiable pattern from the 4-bit test vectors. These patterns are easily replicated for the n-bit case.

Functional Faults	Test Vector (integer)	Test Vector ($X_3X_2X_1X_0$)
F6	0	0000
F1, F2	7	0111
F3, F7, F8	-7	1001
F4, F9	-6	1010
F5, F10	-4	1100

Table 6-12 Generalized functional faults for absolute value and negation.

6.2.4 Behavioral Fault Model

Since *ABS* and *(-)* only operate on integers, the form of their behavioral fault models will be slightly different. One method would be to convert the resulting integer to a *bit_vector*, which could then be corrupted using an error vector and the *XOR* operator. The corrupted *bit_vector* would then have to be converted back to an integer to match the type of the original operation.

Since the purpose of the error vector approach is to simply corrupt the result of the operation, another operator that works directly with integers could just as easily be used. Hence, the *addition operator* (+) is used here instead of the *XOR* to reduce the number of type conversions necessary. An implementation of the behavioral fault model for the *ABS operator* is presented in Figure 6-17. The *negation operator* (-) can also be corrupted using the same method.

```
Z <= (ABS X) + to_integer(x = 0 OR x = 7 OR x = -7 OR
                           x = -6 OR x = -4);
```

Figure 6-17 Behavioral fault model for ABS.

6.2.5 Evaluation of Behavioral Test Vectors

The generalized functional faults from Section 6.2.3 can be readily extrapolated for a larger range of integer values. For examples ABS8 and NEG8, X is declared as an integer with range from -127 to +127. Thus, a synthesis tool will generate hardware with eight bits to represent the 2's complement value of X. The WAVES test vectors for example ABS8 are shown in Figure 6-18.

```
% X          Z      : time ;
00000000 0000000 : 500 ns;
01111111 1111111 : 500 ns;
10000001 1111111 : 500 ns;
10000010 1111110 : 500 ns;
10000100 1111100 : 500 ns;
10001000 1111000 : 500 ns;
10010000 1110000 : 500 ns;
10100000 1100000 : 500 ns;
11000000 1000000 : 500 ns;
```

Figure 6-18 WAVES test vectors for example ABS8.

Example ABS8 was synthesized to a gate level implementation using AutoLogic II. Structure1 is shown in Figure 6-19. Fault simulations using the behavioral test vectors produced a *SSL* gate level fault coverage of $132/132 = 100\%$. The fault coverage graph is shown in Figure 6-20.

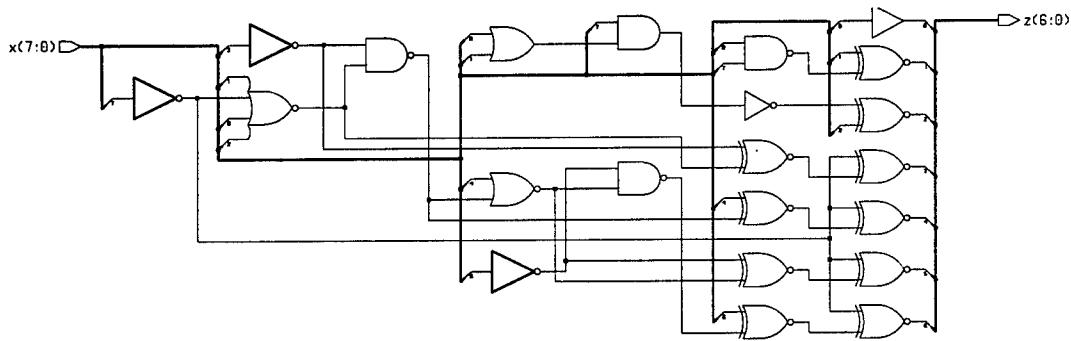


Figure 6-19 Synthesized Structure1 of example ABS8.

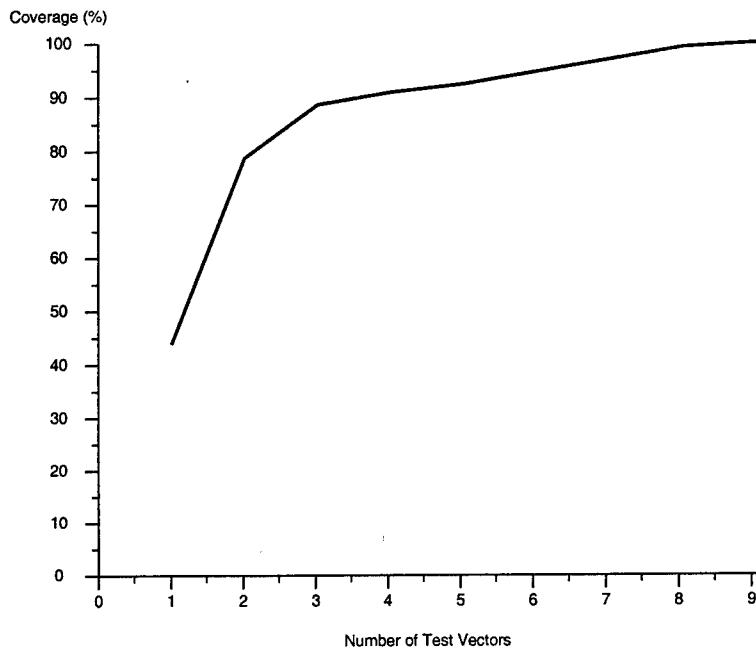


Figure 6-20 Fault coverage for Structure1 of example ABS8.

Due to the common behavioral fault model, the test vectors for example NEG8 are the same as those for ABS8. Example NEG8 was synthesized to the gate level circuit shown in Figure 6-21. According to MIL-STD 883D, the synthesized circuit contains 114 unique

gate level *SSL* faults. Fault simulations using the common behavioral test vectors also result in complete gate level fault coverage.

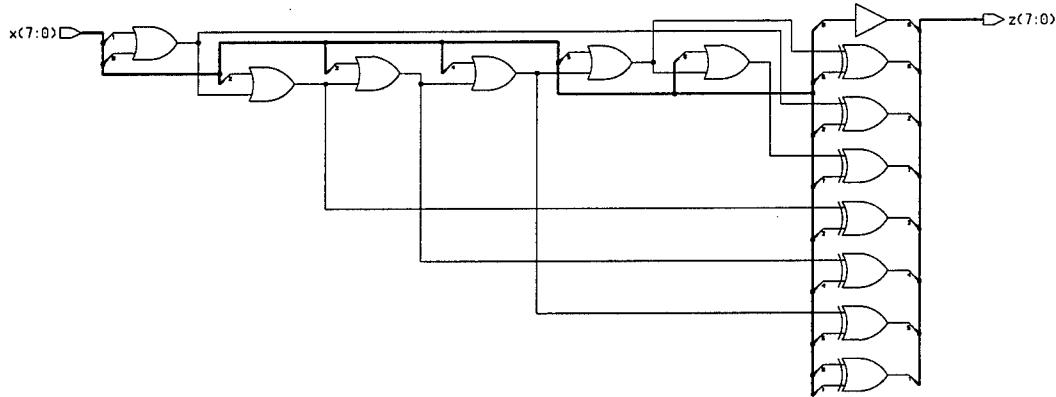


Figure 6-21 Synthesized circuit for example NEG8.

6.3 Conclusions

New behavioral fault models have been developed for the remaining predefined VHDL *operators*. The fault models are based on a functional analysis of the *logical*, *unary*, *multiplying*, and *miscellaneous operators*. Though not necessarily optimal, the new fault models provide complete gate level *SSL* fault coverage over a broad range of hardware implementations.

The new behavioral fault models developed for the *logical operators* (*AND*, *OR*, *NAND*, *NOR*, *XOR*) are based on functional faults that require less than exhaustive testing for all operators except *XOR*. For the special case of *XOR*-only *expressions*, a generalized Bossen algorithm is presented that allows for optimization of test sequences.

The *unary operator* for negation (-) performs a function related to the *miscellaneous operator ABS*; each affects the sign of a 2's complement number. By analyzing each operator's functional faults in tandem with the other, a consistent behavioral fault model was developed. Note that no previous behavioral fault models even address these operators.

Chapter 7

Other Programming Constructs

VHDL also includes other constructs drawn from familiar programming languages. Program *loops*, *functions*, and *procedures* are used in VHDL behavioral descriptions for design simplicity and reuse/repetition of functional blocks. Since any description using these constructs can be rewritten equivalently without them, no additional behavioral faults are implied. Several design examples will be used to demonstrate the interaction between previously defined behavioral faults and these other programming constructs.

7.1 Loops

The VHDL subset, detailed in Appendix D [36], restricts the use of the *loop* statement to only the *for* iteration scheme. The bounds of the discrete range of the *loop* must be specified directly or indirectly as static values belonging to an integer type. Hence, the program *loop* can be expanded or “unrolled” to an equivalent form eliminating the *loop* construct.

7.1.1 A Simple Example

Example SHIFT4u in Figure 7-1 demonstrates the use of program *loops* to perform shifting operations. With control signal *OP* = “01”, a right shift of the unsigned signal *A* is performed by the *I* in 0 to 2 *loop*. Similarly, with *OP* = “10”, a left shift is accomplished via the *I* in 3 downto 1 *loop*. For this example, other values for the control signal *OP* pass signal *A* unchanged.

```
entity shift4u is
    port(
        OP: in std_logic_vector(1 downto 0);
        A: in std_logic_vector(3 downto 0);
        D: out std_logic_vector(3 downto 0)
    );
end shift4u;
```

Figure 7-1 Behavioral description for example SHIFT4u.

```

architecture behave of shift4u is
begin
  process(OP,A)
    variable TMP: std_logic_vector(3 downto 0);
begin
  case OP is
    when "01" =>
      for I in 0 to 2 loop
        TMP(I) := A(I+1);
      end loop;
      TMP(3) := '0';
    when "10" =>
      for I in 3 downto 1 loop
        TMP(I) := A(I-1);
      end loop;
      TMP(0) := '0';
    when others =>
      TMP := A;
  end case;
  D <= TMP;
end process;
end behave;
```

Figure 7-1 Behavioral description for example SHIFT4u.

Due to the directly specified discrete range in each *loop*, they can be readily expanded to a sequence of statements eliminating the *loop* constructs. An expanded version of the *case* statement for example SHIFT4u is shown in Figure 7-2.

```

case OP is
  when "01" =>
    TMP(0) := A(1);
    TMP(1) := A(2);
    TMP(2) := A(3);
    TMP(3) := '0';
  when "10" =>
    TMP(3) := A(2);
    TMP(2) := A(1);
    TMP(1) := A(0);
    TMP(0) := '0';
  when others =>
    TMP := A;
end case;
```

Figure 7-2 Expanded case statement for example SHIFT4u.

The only behavioral faults affecting the expanded example SHIFT4u are the control faults for the *case* statement. The behavioral faults and resulting test vectors are shown in Table 7-1. Combining don't cares produces the behavioral test vectors in Figure 7-3.

Behavioral Fault	Corrupted Clause	Test Vectors (OP A)
WHEN-00-CORRUPT (by WHEN-01)(OR)	$\text{TMP}(0) := \text{A}(0) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(1) := \text{A}(1) \text{ OR } \mathbf{A}(2)$ $\text{TMP}(2) := \text{A}(2) \text{ OR } \mathbf{A}(3)$ $\text{TMP}(3) := \text{A}(3) \text{ OR } '0'$	00 1010 00 X10X
WHEN-00-CORRUPT (by WHEN-01)(AND)	$\text{TMP}(0) := \text{A}(0) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(1) := \text{A}(1) \text{ AND } \mathbf{A}(2)$ $\text{TMP}(2) := \text{A}(2) \text{ AND } \mathbf{A}(3)$ $\text{TMP}(3) := \text{A}(3) \text{ AND } '0'$	00 0101 00 101X
WHEN-00-CORRUPT (by WHEN-10)(OR)	$\text{TMP}(0) := \text{A}(0) \text{ OR } '0'$ $\text{TMP}(1) := \text{A}(1) \text{ OR } \mathbf{A}(0)$ $\text{TMP}(2) := \text{A}(2) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(3) := \text{A}(3) \text{ OR } \mathbf{A}(2)$	00 0101 00 X01X
WHEN-00-CORRUPT (by WHEN-10)(AND)	$\text{TMP}(0) := \text{A}(0) \text{ AND } '0'$ $\text{TMP}(1) := \text{A}(1) \text{ AND } \mathbf{A}(0)$ $\text{TMP}(2) := \text{A}(2) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(3) := \text{A}(3) \text{ AND } \mathbf{A}(2)$	00 1010 00 X101
WHEN-01-CORRUPT (by WHEN-00)(OR)	$\text{TMP}(0) := \text{A}(1) \text{ OR } \mathbf{A}(0)$ $\text{TMP}(1) := \text{A}(2) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(2) := \text{A}(3) \text{ OR } \mathbf{A}(2)$ $\text{TMP}(3) := '0' \text{ OR } \mathbf{A}(3)$	01 0101 01 101X
WHEN-01-CORRUPT (by WHEN-00)(AND)	$\text{TMP}(0) := \text{A}(1) \text{ AND } \mathbf{A}(0)$ $\text{TMP}(1) := \text{A}(2) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(2) := \text{A}(3) \text{ AND } \mathbf{A}(2)$ $\text{TMP}(3) := '0' \text{ AND } \mathbf{A}(3)$	01 1010 01 X10X
WHEN-01-CORRUPT (by WHEN-11)(OR)	$\text{TMP}(0) := \text{A}(1) \text{ OR } \mathbf{A}(0)$ $\text{TMP}(1) := \text{A}(2) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(2) := \text{A}(3) \text{ OR } \mathbf{A}(2)$ $\text{TMP}(3) := '0' \text{ OR } \mathbf{A}(3)$	01 0101 01 101X
WHEN-01-CORRUPT (by WHEN-11)(AND)	$\text{TMP}(0) := \text{A}(1) \text{ AND } \mathbf{A}(0)$ $\text{TMP}(1) := \text{A}(2) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(2) := \text{A}(3) \text{ AND } \mathbf{A}(2)$ $\text{TMP}(3) := '0' \text{ AND } \mathbf{A}(3)$	01 1010 01 X10X

Table 7-1 Behavioral faults for example SHIFT4u.

Behavioral Fault	Corrupted Clause	Test Vectors (OP A)
WHEN-10-CORRUPT (by WHEN-00)(OR)	$\text{TMP}(0) := '0' \text{ OR } \mathbf{A}(0)$ $\text{TMP}(1) := \mathbf{A}(0) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(2) := \mathbf{A}(1) \text{ OR } \mathbf{A}(2)$ $\text{TMP}(3) := \mathbf{A}(2) \text{ OR } \mathbf{A}(3)$	10 1010 10 X101
WHEN-10-CORRUPT (by WHEN-00)(AND)	$\text{TMP}(0) := '0' \text{ AND } \mathbf{A}(0)$ $\text{TMP}(1) := \mathbf{A}(0) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(2) := \mathbf{A}(1) \text{ AND } \mathbf{A}(2)$ $\text{TMP}(3) := \mathbf{A}(2) \text{ AND } \mathbf{A}(3)$	10 0101 10 X01X
WHEN-10-CORRUPT (by WHEN-11)(OR)	$\text{TMP}(0) := '0' \text{ OR } \mathbf{A}(0)$ $\text{TMP}(1) := \mathbf{A}(0) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(2) := \mathbf{A}(1) \text{ OR } \mathbf{A}(2)$ $\text{TMP}(3) := \mathbf{A}(2) \text{ OR } \mathbf{A}(3)$	10 1010 10 X101
WHEN-10-CORRUPT (by WHEN-11)(AND)	$\text{TMP}(0) := '0' \text{ AND } \mathbf{A}(0)$ $\text{TMP}(1) := \mathbf{A}(0) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(2) := \mathbf{A}(1) \text{ AND } \mathbf{A}(2)$ $\text{TMP}(3) := \mathbf{A}(2) \text{ AND } \mathbf{A}(3)$	10 0101 10 X01X
WHEN-11-CORRUPT (by WHEN-01)(OR)	$\text{TMP}(0) := \mathbf{A}(0) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(1) := \mathbf{A}(1) \text{ OR } \mathbf{A}(2)$ $\text{TMP}(2) := \mathbf{A}(2) \text{ OR } \mathbf{A}(3)$ $\text{TMP}(3) := \mathbf{A}(3) \text{ OR } '0'$	11 1010 11 X10X
WHEN-11-CORRUPT (by WHEN-01)(AND)	$\text{TMP}(0) := \mathbf{A}(0) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(1) := \mathbf{A}(1) \text{ AND } \mathbf{A}(2)$ $\text{TMP}(2) := \mathbf{A}(2) \text{ AND } \mathbf{A}(3)$ $\text{TMP}(3) := \mathbf{A}(3) \text{ AND } '0'$	11 0101 11 101X
WHEN-11-CORRUPT (by WHEN-10)(OR)	$\text{TMP}(0) := \mathbf{A}(0) \text{ OR } '0'$ $\text{TMP}(1) := \mathbf{A}(1) \text{ OR } \mathbf{A}(0)$ $\text{TMP}(2) := \mathbf{A}(2) \text{ OR } \mathbf{A}(1)$ $\text{TMP}(3) := \mathbf{A}(3) \text{ OR } \mathbf{A}(2)$	11 0101 11 X01X
WHEN-11-CORRUPT (by WHEN-10)(AND)	$\text{TMP}(0) := \mathbf{A}(0) \text{ AND } '0'$ $\text{TMP}(1) := \mathbf{A}(1) \text{ AND } \mathbf{A}(0)$ $\text{TMP}(2) := \mathbf{A}(2) \text{ AND } \mathbf{A}(1)$ $\text{TMP}(3) := \mathbf{A}(3) \text{ AND } \mathbf{A}(2)$	11 1010 11 X101

Table 7-1 Behavioral faults for example SHIFT4u.

```
%OP A D : time;
00 0101 0101 : 500 ns;
00 1010 1010 : 500 ns;
01 0101 0010 : 500 ns;
01 1010 0101 : 500 ns;
10 0101 1010 : 500 ns;
10 1010 0100 : 500 ns;
11 0101 0101 : 500 ns;
11 1010 1010 : 500 ns;
```

Figure 7-3 WAVES test vectors for example SHIFT4u.

Example SHIFT4u from Figure 7-1 was synthesized to the gate level Structure1 shown in Figure 7-4. Fault simulations were performed using the behavioral test vectors from Figure 7-3. The resulting SSL gate level fault coverage of $90/90 = 100\%$ is shown in Figure 7-5. An alternate synthesis tool and target architecture was next used to produce Structure2 for example SHIFT4u. Fault simulations using the behavioral test vectors from Figure 7-3 resulted in a SSL gate level fault coverage of $112/112 = 100\%$.

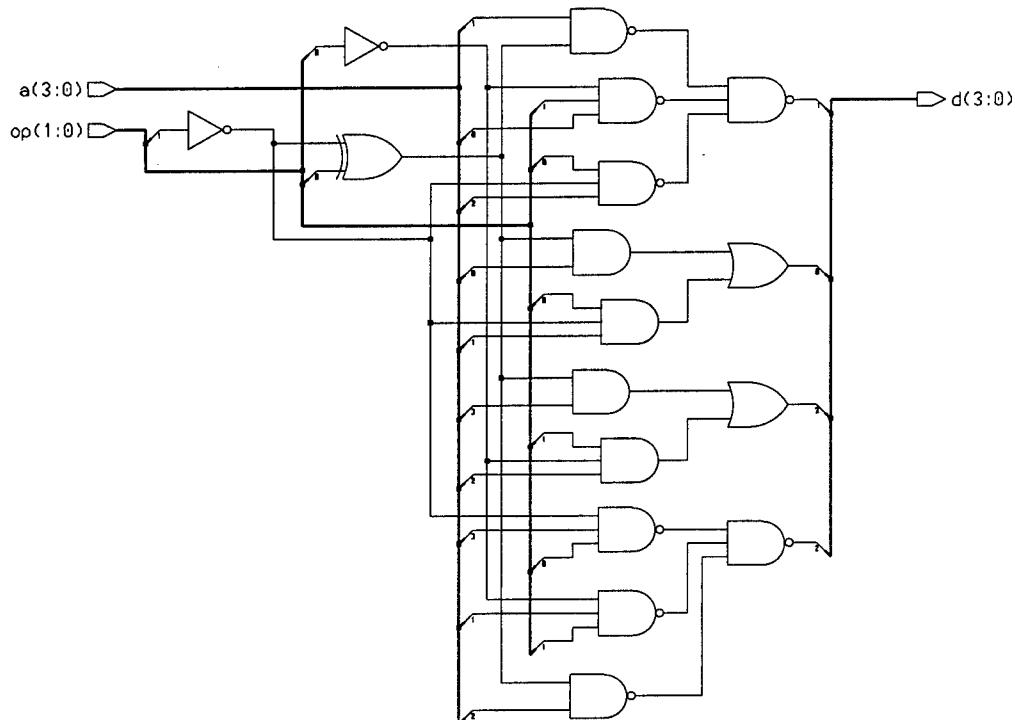


Figure 7-4 Synthesized Structure1 for example SHIFT4u.

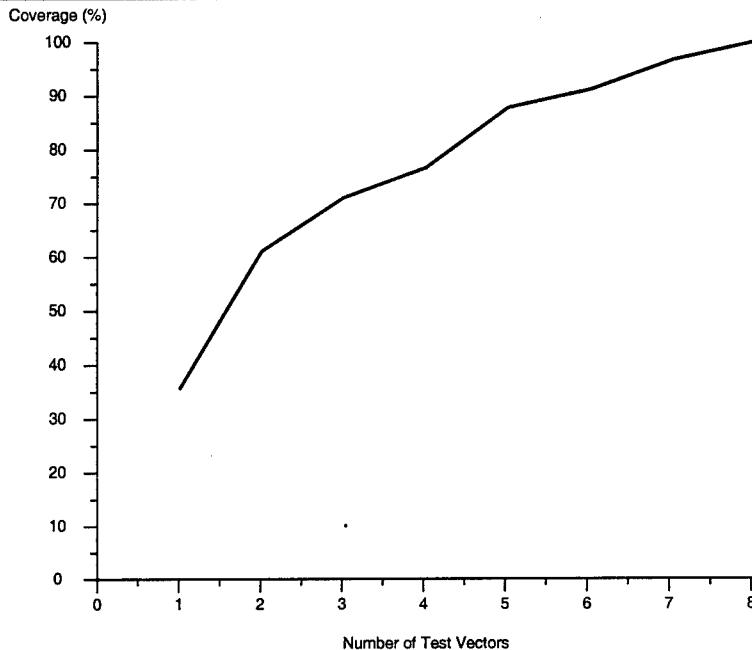


Figure 7-5 Fault coverage for Structure1 of example SHIFT4u.

7.1.2 Comparison with Previous Fault Models

The only previous behavioral fault model to address the *loop* construct was that proposed by Riesgo and Uceda [60]. As part of faults on *expressions*, the *for_in_loop* construct was faulted by the *index* controlling the *loop* changing its range from the *minimum* to the *maximum+1* and from the *minimum-1* to the *maximum*. As can be seen from example SHIFT4u, such faults would produce signals that do not even exist in the synthesized hardware, $A(-1)$ and $A(3)$. Thus, their proposed fault model is more a software mutation, than hardware oriented as they claim.

The restrictions placed on the *loop* construct by the VHDL synthesis subset imply that all such *loops* can be readily expanded. Since this expansion or “unrolling” eliminates the *loops* from the behavioral description, no additional behavioral faults are introduced. The next section will show that the behavior of *functions* and *procedures* is much the same.

7.2 Functions and Procedures

Much like program *loops*, *functions* and *procedures* are used mainly as a convenience for ease of programming. In general, any VHDL code written with *functions* and *procedures* can be mapped to the same hardware as equivalent code without *functions* or *procedures* [10]. Since the VHDL behavioral descriptions can be expanded to eliminate these

programming constructs, no additional behavioral faults are introduced by the use of *functions* or *procedures*.

7.2.1 Example ADD4fn

Example ADD4fn in Figure 7-6 describes a 4-bit *ripple carry* adder using *functions* to perform the *sum* and *carry* operations. The functions FA_S and FA_C are shown in Figure 7-7.

```

process(A,B,CIN)
  variable CARRY: std_logic_vector(4 downto 0);
  variable SUM : std_logic_vector(3 downto 0);
begin
  CARRY(0) := CIN;
  for I in 0 to 3 loop
    SUM(I) := FA_S(A(I), B(I), CARRY(I));
    CARRY(I+1) := FA_C(A(I), B(I), CARRY(I));
  end loop;
  S <= SUM;
  COUT <= CARRY(4);
end process;

```

Figure 7-6 Behavioral description for example ADD4fn.

```

function FA_S (AIN, BIN, CIN: std_logic) return
  std_logic is
begin
  return AIN xor BIN xor CIN;
end FA_S;

function FA_C (AIN, BIN, CIN: std_logic) return
  std_logic is
begin
  return (AIN and BIN) or (AIN and CIN) or
    (BIN and CIN);
end FA_C;

```

Figure 7-7 Functions for example ADD4fn.

The *loop* and *function* programming constructs can be eliminated from the behavioral description by expanding the *loop* and replacing the *function* call with its returned *expression*. The results of this expansion and substitution are shown in Figure 7-8. Since the VHDL behavioral descriptions in Figure 7-6 and Figure 7-8 synthesize to the same gate

level structures, no additional behavioral faults have been introduced by the use of *functions*.

```

begin
    CARRY(0) := CIN;
    SUM(0) := A(0) xor B(0) xor CARRY(0);
    CARRY(1) := (A(0) and B(0)) or (A(0) and CARRY(0)) or
                (B(0) and CARRY(0));
    SUM(1) := A(1) xor B(1) xor CARRY(1);
    CARRY(2) := (A(1) and B(1)) or (A(1) and CARRY(1)) or
                (B(1) and CARRY(1));
    SUM(2) := A(2) xor B(2) xor CARRY(2);
    CARRY(3) := (A(2) and B(2)) or (A(2) and CARRY(2)) or
                (B(2) and CARRY(2));
    SUM(3) := A(3) xor B(3) xor CARRY(3);
    CARRY(4) := (A(3) and B(3)) or (A(3) and CARRY(3)) or
                (B(3) and CARRY(3));
    S <= SUM;
    COUT <= CARRY(4);
end process;

```

Figure 7-8 Expanded behavioral description for example ADD4fn.

7.2.2 Example ADD4pr

The 4-bit *ripple carry* adder can be equivalently written using a *procedure* as shown in Figure 7-9. The *procedure* FA is shown in Figure 7-10.

```

process(A,B,CIN)
    variable CARRY: std_logic_vector(4 downto 0);
    variable SUM : std_logic_vector(3 downto 0);
begin
    CARRY(0) := CIN;
    for I in 0 to 3 loop
        FA(A(I), B(I), CARRY(I), SUM(I), CARRY(I+1));
    end loop;
    S <= SUM;
    COUT <= CARRY(4);
end process;

```

Figure 7-9 Behavioral description for example ADD4pr.

```
procedure FA (AIN, BIN, CIN: in std_logic;
              SOUT, COUT: out std_logic) is
begin
    SOUT := AIN xor BIN xor CIN;
    COUT := (AIN and BIN) or (AIN and CIN) or
            (BIN and CIN);
end FA;
```

Figure 7-10 Procedure FA for example ADD4pr.

Since the VHDL behavioral description ADD4pr in Figure 7-9 synthesizes to the same gate level structures as the examples in Figure 7-6 and Figure 7-8, no additional behavioral faults have been introduced by the use of *procedures*.

7.3 Conclusions

VHDL includes other constructs drawn from familiar programming languages. Program *loops*, *functions*, and *procedures* are used in VHDL behavioral descriptions for design simplicity and reuse/repetition of functional blocks. Since any description using these constructs can be rewritten equivalently without them, no additional behavioral faults are implied. Several design examples were used to demonstrate the interaction between previously defined behavioral faults and these other programming constructs.

Chapter 8

Comprehensive Examples

Two comprehensive examples have been chosen to demonstrate the gate level fault coverage of the new behavioral fault models. The first is an arithmetic logic unit (ALU) which performs selected functions on data inputs. The second example is a single error correcting circuit used in fault tolerant applications. Other obvious combinational logic examples such as a multiplexer or a magnitude comparator do not need to be investigated here due to their detailed analysis as part of the development of the fault models for the *if* statement and *relational operators*.

Application of the behavioral fault models to each of the comprehensive examples results in a set of test vectors necessary to detect the behavioral faults. These test vector sets are then applied to synthesized gate level implementations of the behavioral descriptions. Resulting gate level fault coverage is evaluated to determine the effectiveness of the behavioral fault models.

8.1 Arithmetic Logic Unit

The ALU design for this example was created using the LogicLib generator from the Mentor Graphics design tools. The type was selected as an ALU2901 which performs eight arithmetic and logic functions. Data widths of 4- and 8-bits will be evaluated.

8.1.1 Example ALU4wc

The generator parameters and the resulting entity description for a 4-bit ALU with both *carry-in* and *carry-out* are shown in Figure 8-1. The architecture description for

```
-- Written by LL_to_VHDL at Mon Jun  8 12:23:29 1998
-- Parameterized Generator Specification to VHDL Code
-- LogicLib generator called: ARITHMETIC
-- Passed Parameters are:
--     type = ALU2901
--     W = 4
--     carryin = YES
--     carryout = YES
```

Figure 8-1 Entity description for example ALU4wc.

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;
-- alu4wc Entity Description
entity alu4wc is
    port(
        OP: in std_logic_vector(2 downto 0);
        A: in std_logic_vector(3 downto 0);
        B: in std_logic_vector(3 downto 0);
        CIN: in std_logic;
        COUT: out std_logic;
        D: out std_logic_vector(3 downto 0)
    );
end alu4wc;

```

Figure 8-1 Entity description for example ALU4wc.

example ALU4wc is shown in Figure 8-2. In the first section, variables are declared and initialized consistent with arithmetic operations involving a *carry-in* and a *carry-out*. Next, two *case* statements determine the appropriate operation to be performed. Lastly, the outputs are assigned based on whether the operation performed was *arithmetic* or *logic*.

```

architecture behave of alu4wc is
begin
    ARITHMETIC_Process: process(A,B,CIN,OP)
        variable operand1: std_logic_vector(4 downto 0);
        variable operand2: std_logic_vector(4 downto 0);
        variable a_ext: std_logic_vector(4 downto 0);
        variable b_ext: std_logic_vector(4 downto 0);
        variable not_a_ext: std_logic_vector(4 downto 0);
        variable not_b_ext: std_logic_vector(4 downto 0);
        variable carry_ext: std_logic_vector(1 downto 0);
        variable logic_out: std_logic_vector(3 downto 0);
        variable arith_out: std_logic_vector(4 downto 0);
    begin
        -- zero extend inputs to include carry bit
        a_ext := '0' & A;
        b_ext := '0' & B;
        not_a_ext := '0' & not A;
        not_b_ext := '0' & not B;
        carry_ext := '0' & CIN;

```

Figure 8-2 Architecture description for example ALU4wc.

```
-- ALU2901

-----
-- Logical Functions --
-----

case OP is
    when "011" =>
        logic_out := A or B;
    when "100" =>
        logic_out := A and B;
    when "101" =>
        logic_out := (not A) and B;
    when "110" =>
        logic_out := A xor B;
    when "111" =>
        logic_out := not (A xor B);
    when others =>
        logic_out := (OTHERS => 'X');
end case;

-----
-- Arithmetic Functions --
-----

case OP is
    -- Arithmetic operations
    when "000" =>
        operand1 := a_ext;
        operand2 := b_ext;
    when "001" =>
        operand1 := not_a_ext;
        operand2 := b_ext;
    when "010" =>
        operand1 := a_ext;
        operand2 := not_b_ext;
    when others =>
        operand1 := (OTHERS => 'X');
        operand2 := (OTHERS => 'X');
end case;

arith_out := operand1 + operand2 + carry_ext;
```

Figure 8-2 Architecture description for example ALU4wc.

```

-- assign output
if (OP(2) = '1' or (OP(1) = '1' and OP(0) = '1'))
then
    D <= logic_out;
    COUT <= 'X';
else
    D <= arith_out(3 downto 0);
    COUT <= arith_out(4);
end if;

end process ARITHMETIC_Process;
end behave;

```

Figure 8-2 Architecture description for example ALU4wc.

8.1.1.1 Faults on Logical Operators

Within the **Logical Functions** *case* statement, the variable *logic_out* is determined by combining the signals *A*, *B* using the *logical operators AND, OR, and XOR*. For the first logical expression, when *OP* = 011:

```
logic_out := A OR B;
```

From the behavioral fault models for *logical operators*, the three test vectors (*AB*) necessary for an *OR* operator are 00, 01, and 10. Since all signals are in fact four bits wide, these tests expand to produce the behavioral test vectors shown in Table 8-1.

Expression	OP	A	B	logic_out
A OR B	011	0000	0000	0000
		0000	1111	1111
		1111	0000	1111

Table 8-1 Behavioral test vectors for OR operator.

The behavioral test vectors for the remaining **Logical Functions** are determined in a similar manner. The *logical operator AND* also requires three tests: 01, 10 and 11. Next, the *XOR* operator requires a complete set of four test vectors: 00, 01, 10, and 11. Like the *OR* operation, these vectors also expand to four bits for each *operator*. The resulting behavioral test vectors are shown in Table 8-2.

Expression	OP	A	B	logic_out
A AND B	100	0000	1111	0000
		1111	0000	0000
		1111	1111	1111
(not A) AND B	101	1111	1111	0000
		0000	0000	0000
		0000	1111	1111
A XOR B	110	0000	0000	0000
		0000	1111	1111
		1111	0000	1111
		1111	1111	0000
not (A XOR B)	111	0000	0000	1111
		0000	1111	0000
		1111	0000	0000
		1111	1111	1111

Table 8-2 Behavioral test vectors for remaining Logical Functions.

8.1.1.2 Faults on Arithmetic Operators

The *ADD with carry* operation performed in the **Arithmetic Functions** section determines the variable *arith_out* as follows:

```
arith_out := operand1 + operand2 + carry_ext;
```

Assuming a simple *ripple carry* implementation, the behavioral test vectors for the 4-bit addition come directly from Table 5-6 and are shown here in Table 8-3.

Due to the *case* statement for **Arithmetic Functions**, there are three different ways to form *operand1* and *operand2*. The resulting possible test vectors are shown in Table 8-4. Only one input combination is required for each **Test #**, though using all possibilities would not be incorrect, just redundant. As with previous examples, the *control faults* can provide the necessary insight for selecting a reduced set of behavioral test vectors.

Phase	Test #	operand1	operand2	carry_ext	arith_out
I	1	0000	0000	0	0 0000
	2	0000	1111	0	0 1111
	3	1111	0000	0	0 1111
	4	1111	1111	1	1 1111
II	5	0000	1111	1	1 0000
	6	1111	0000	1	1 0000
III	7	0101	0101	0	0 1010
	8	1010	1010	1	1 0101

Table 8-3 Behavioral tests for 4-bit ADD with carry.

OP	Phase	Test #	A	B	CIN	arith_out
000	I	1	0000	0000	0	0 0000
		2	0000	1111	0	0 1111
		3	1111	0000	0	0 1111
		4	1111	1111	1	1 1111
	II	5	0000	1111	1	1 0000
		6	1111	0000	1	1 0000
	III	7	0101	0101	0	0 1010
		8	1010	1010	1	1 0101
001	I	1	1111	0000	0	0 0000
		2	1111	1111	0	0 1111
		3	0000	0000	0	0 1111
		4	0000	1111	1	1 1111
	II	5	1111	1111	1	1 0000
		6	0000	0000	1	1 0000
	III	7	1010	0101	0	0 1010
		8	0101	1010	1	1 0101

Table 8-4 Possible test vectors for Arithmetic Functions.

OP	Phase	Test #	A	B	CIN	arith_out
010	I	1	0000	1111	0	0 0000
		2	0000	0000	0	0 1111
		3	1111	1111	0	0 1111
		4	1111	0000	1	1 1111
	II	5	0000	0000	1	1 0000
		6	1111	1111	1	1 0000
	III	7	0101	1010	0	0 1010
		8	1010	0101	1	1 0101

Table 8-4 Possible test vectors for Arithmetic Functions.

8.1.1.3 Control Faults

The control fault model states that each *clause* of an *if* or *case* statement is corrupted by other *clauses* that are logically adjacent. For the **Logical Functions** *case* statement, the first clause, *WHEN-011*, can be corrupted by either *WHEN-001* or *WHEN-010*. Both of these cases fall under the *others clause* and, hence, cause no corruption due to the don't care values. The third possible corruption is caused by the *WHEN-111 clause*. For example, the control fault *WHEN-011 CORRUPT (by WHEN-111)(OR)* produces the corrupted clause shown below.

```
logic_out := (A OR B) OR (not (A XOR B));
```

Test vector generation rules specify that $(A \text{ OR } B)$ be set to 0, while $(\text{not } (A \text{ XOR } B))$ is set to 1. Checking the previously determined test vectors for *logical operators* from Table 8-1 and Table 8-2, the test vector with $A = 0000$ and $B = 0000$ meets these requirements and, hence, covers this control fault.

The control faults for all the **Logical Functions** are shown in Table 8-5. The faults are grouped according to the *corrupted clauses*. Each *corrupting clause* includes both an *OR-fault* and an *AND-fault*. The test generation rules provide requirements that produce the appropriate test vector.

Corrupted Clause	Corrupting Clause	Fault	Test Requirement	Test Vector (<i>OPAB</i>)
011	001	NONE		
	010	NONE		
	111	OR	A OR B = 0 not (A XOR B) = 1	011 0000 0000
		AND	A OR B = 1 not (A XOR B) = 0	011 0000 1111
100	000	NONE		
	101	OR	A AND B = 0 (not A) AND B = 1	100 0000 1111
		AND	A AND B = 1 (not A) AND B = 0	100 1111 1111
	110	OR	A AND B = 0 A XOR B = 1	100 0000 1111
		AND	A AND B = 1 A XOR B = 0	100 1111 1111
101	001	NONE		
	100	OR	(not A) AND B = 0 A AND B = 1	101 1111 1111
		AND	(not A) AND B = 1 A AND B = 0	101 0000 1111
	111	OR	(not A) AND B = 0 not (A XOR B) = 1	101 1111 1111
		AND	(not A) AND B = 1 not (A XOR B) = 0	101 0000 1111
110	010	NONE		
	100	OR	A XOR B = 0 A AND B = 1	110 1111 1111
		AND	A XOR B = 1 A AND B = 0	110 0000 1111

Table 8-5 Control faults for Logical Functions.

Corrupted Clause	Corrupting Clause	Fault	Test Requirement	Test Vector (<i>O P A B</i>)
110	111	OR	A XOR B = 0 not (A XOR B) = 1	110 0000 0000
		AND	A XOR B = 1 not (A XOR B) = 0	110 1111 0000
111	011	OR	not (A XOR B) = 0 A OR B = 1	111 0000 1111
		AND	not (A XOR B) = 1 A OR B = 0	111 0000 0000
111	101	OR	not (A XOR B) = 0 (not A) AND B = 1	111 0000 1111
		AND	not (A XOR B) = 1 (not A) AND B = 0	111 1111 1111
111	110	OR	not (A XOR B) = 0 A XOR B = 1	111 1111 0000
		AND	not (A XOR B) = 1 A XOR B = 0	111 0000 0000

Table 8-5 Control faults for Logical Functions.

The control faults for the **Arithmetic Functions** are formed in the same manner. The resulting faults and their test vectors are shown in Table 8-6.

Corrupted Clause	Corrupting Clause	Fault	Test Requirement	Test Vector (<i>O P A B</i>)
000	001	OR	a_ext = 0 not_a_ext = 1	000 0000 XXXX
		AND	a_ext = 1 not_a_ext = 0	000 1111 XXXX
000	010	OR	b_ext = 0 not_b_ext = 1	000 XXXX 0000
		AND	b_ext = 0 not_b_ext = 1	000 XXXX 1111
100	NONE			

Table 8-6 Control faults for Arithmetic Functions.

Corrupted Clause	Corrupting Clause	Fault	Test Requirement	Test Vector ($OP\ A\ B$)
001	000	OR	not_a_ext = 0 a_ext = 1	001 1111 xxxx
		AND	not_a_ext = 1 a_ext = 0	001 0000 xxxx
	011	NONE		
	101	NONE		
010	000	OR	not_b_ext = 0 b_ext = 1	010 xxxx 1111
		AND	not_b_ext = 1 b_ext = 0	010 xxxx 0000
	011	NONE		
	110	NONE		

Table 8-6 Control faults for Arithmetic Functions.

The final control faults affect the *if* statement that assigns the outputs. The *then* and *else* clauses determine whether the function performed was *logic* or *arithmetic*, respectively. The logical adjacencies among the clauses are illustrated in Figure 8-3; *logic* functions corresponding to the *then* clause are shaded. For example, the *logic* function for $OP = 011$ can be corrupted by the *arithmetic* functions with $OP = 001$ and $OP = 010$.

		OP(1) OP(0)			
		00	01	11	10
OP(2)	0	000	001	011	010
	1	100	101	111	110
OP					

Figure 8-3 Logical adjacencies among clauses.

THEN-CORRUPT faults cause *logic* functions to be corrupted by *arithmetic* functions. As an example, the control fault *100-CORRUPT* (by 000)(OR) produces the following corrupted assignment statements:

```
D <= logic_out OR arith_out(3 downto 0);
COUT <= 'X' OR arith_out(4);
```

Candidate test vectors for $OP = 100$ with corresponding values for $logic_out$ are found in Table 8-2. Corrupting values of $arith_out(3 \text{ downto } 0)$ for $OP = 000$ can be found in Table 8-4. Comparing these values shows that an appropriate test vector is formed by $A = 0000$ and $B = 1111$ with $CIN = 0$. Note that a *logic* operation would normally leave CIN as a don't care, however, corruption by an *arithmetic* operation requires specification of this value. *THEN-CORRUPT* faults and their resulting test vectors are shown in Table 8-7.

Corrupted Clause	Corrupting Clause	Fault	Test Requirement	Test Vector ($OP\ A\ B\ CIN$)
011	001	OR	$logic_out = 0$ $arith_out = 1$	011 0000 0000 0
		AND	$logic_out = 1$ $arith_out = 0$	011 1111 0000 0
	010	OR	$logic_out = 0$ $arith_out = 1$	011 0000 0000 0
		AND	$logic_out = 1$ $arith_out = 0$	011 0000 1111 0
100	000	OR	$logic_out = 0$ $arith_out = 1$	100 0000 1111 0
		AND	$logic_out = 1$ $arith_out = 0$	100 0101 0101 0 100 1010 1010 1
	001	OR	$logic_out = 0$ $arith_out = 1$	101 1111 1111 0
101	001	AND	$logic_out = 1$ $arith_out = 0$	101 1010 0101 0 101 0101 1010 1
		OR	$logic_out = 0$ $arith_out = 1$	110 0000 0000 0
110	010	AND	$logic_out = 1$ $arith_out = 0$	110 0000 1111 0

Table 8-7 THEN-CORRUPT control faults.

ELSE-CORRUPT faults are formed in a similar manner resulting in corruption of an *arithmetic* function by a *logic* function. *ELSE-CORRUPT* faults are shown in Table 8-8.

Corrupted Clause	Corrupting Clause	Fault	Test Requirement	Test Vector (<i>OP A B CIN</i>)
000	100	OR	arith_out = 0 logic_out = 1	000 0101 0101 0 000 1010 1010 1
		AND	arith_out = 1 logic_out = 0	000 0101 0101 0 000 1010 1010 1
001	011	OR	arith_out = 0 logic_out = 1	001 1111 0000 0
		AND	arith_out = 1 logic_out = 0	001 0000 0000 0
010	101	OR	arith_out = 0 logic_out = 1	001 1010 0101 0 001 0101 1010 1
		AND	arith_out = 1 logic_out = 0	001 1010 0101 0 001 0101 1010 1
	011	OR	arith_out = 0 logic_out = 1	010 1111 1111 1
		AND	arith_out = 1 logic_out = 0	010 0000 0000 0
	110	OR	arith_out = 0 logic_out = 1	010 0000 1111 0
		AND	arith_out = 1 logic_out = 0	010 0000 0000 0

Table 8-8 ELSE-CORRUPT control faults.

8.1.1.4 Final Behavioral Test Vector Set

Combining the control faults for **Logical Functions** from Table 8-5 and the **THEN-CORRUPT** control faults from Table 8-7 with the tests for **Logical Functions** from Table 8-1 and Table 8-2 will produce a final set of behavioral test vectors for $OP = 011$ through $OP = 111$. Further optimization is possible by noting that the two test vectors required for the control fault **100-CORRUPT** (by 000)(AND) provide coverage for the $A = 1, B = 1$ fault to the AND operator for $OP = 100$. A similar optimization applies to the control fault **101-CORRUPT** (by 001)(AND). The resulting behavioral test vectors and covered control faults are shown in Table 8-9.

Expression	Test Vector (OP A B CIN)	Control Faults
A OR B	011 0000 0000 0	011 by 001 (OR), 011 by 010 (OR), 011 by 111 (OR)
	011 0000 1111 0	011 by 010 (AND), 011 by 111 (AND)
	011 1111 0000 0	011 by 001 (AND)
A AND B	100 0000 1111 0	100 by 000 (OR), 100 by 101 (OR), 100 by 110 (OR)
	100 1111 0000 X	
	100 0101 0101 0 100 1010 1010 1	100 by 000 (AND), 100 by 101 (AND), 100 by 110 (AND)
(not A) AND B	101 1111 1111 0	101 by 001 (OR), 101 by 100 (OR), 101 by 111 (OR)
	101 0000 0000 X	
	101 1010 0101 0 101 0101 1010 1	101 by 001 (AND), 101 by 100 (AND), 101 by 111 (AND)
A XOR B	110 0000 0000 0	110 by 010 (OR), 110 by 111 (OR)
	110 0000 1111 0	110 by 010 (AND), 110 by 100 (AND)
	110 1111 0000 X	110 by 111 (AND)
	110 1111 1111 X	110 by 100 (OR)
not (A XOR B)	111 0000 0000 X	111 by 011 (AND), 111 by 110 (AND)
	111 0000 1111 X	111 by 011 (OR), 111 by 101 (OR)
	111 1111 0000 X	111 by 110 (OR)
	111 1111 1111 X	111 by 101 (AND)

Table 8-9 Final behavioral test vectors for Logical Functions.

Likewise, the control faults for **Arithmetic Functions** from Table 8-6 and the *ELSE-CORRUPT* control faults from Table 8-8 are combined with the possible test vectors for **Arithmetic Functions** from Table 8-4. The only behavioral tests, for the 4-bit *ADD with carry*, not specified by control faults are **Test 5** and **Test 6**. Test vectors to cover this Phase were selected from $OP = 010$ simply to balance the number of test vectors in each group. The resulting behavioral test vectors and covered control faults are shown in Table 8-10.

Phase	Test #	Test Vector (OP A B CIN)	Control Faults
I	1	000 0000 0000 0	000 by 001 (OR), 000 by 010 (OR)
		001 1111 0000 0	001 by 000 (OR), 001 by 011 (OR)
		010 0000 1111 0	010 by 000 (OR), 010 by 011 (OR), 010 by 110 (OR)
	2	010 0000 0000 0	010 by 000 (AND), 010 by 011 (AND), 010 by 110 (AND)
	3	001 0000 0000 0	001 by 000 (AND), 001 by 011 (AND)
	4	000 1111 1111 1	000 by 001 (AND), 000 by 010 (AND)
II	5	010 0000 0000 1	
	6	010 1111 1111 1	
III	7	000 0101 0101 0	000 by 100 (OR), 000 by 100 (AND)
	8	000 1010 1010 1	
	7	001 1010 0101 0	001 by 101 (OR), 000 by 101 (AND)
	8	001 0101 1010 1	

Table 8-10 Final behavioral test vectors for Arithmetic Functions.

Hence, a final set of 31 test vectors has been formed by application of the new behavioral fault models. The behavioral test vectors and resulting outputs are presented in WAVES format in Figure 8-4.

```
% OP A B CIN COUT D : time;
000 0000 0000 0 0 0000 : 500 ns;
000 1111 1111 1 1 1111 : 500 ns;
000 0101 0101 0 0 1010 : 500 ns;
000 1010 1010 1 1 0101 : 500 ns;
%
001 1111 0000 0 0 0000 : 500 ns;
001 0000 0000 0 0 1111 : 500 ns;
001 1010 0101 0 0 1010 : 500 ns;
001 0101 1010 1 1 0101 : 500 ns;
%
```

Figure 8-4 WAVES test vectors for example ALU4wc.

```

010 0000 1111 0 0 0000 : 500 ns;
010 0000 0000 0 0 1111 : 500 ns;
010 0000 0000 1 1 0000 : 500 ns;
010 1111 1111 1 1 0000 : 500 ns;
%
011 0000 0000 0 X 0000 : 500 ns;
011 0000 1111 0 X 1111 : 500 ns;
011 1111 0000 0 X 1111 : 500 ns;
%
100 0000 1111 0 X 0000 : 500 ns;
100 1111 0000 X X 0000 : 500 ns;
100 0101 0101 0 X 0101 : 500 ns;
100 1010 1010 1 X 1010 : 500 ns;
%
101 1111 1111 0 X 0000 : 500 ns;
101 0000 0000 X X 0000 : 500 ns;
101 1010 0101 0 X 0101 : 500 ns;
101 0101 1010 1 X 1010 : 500 ns;
%
110 0000 0000 0 X 0000 : 500 ns;
110 0000 1111 0 X 1111 : 500 ns;
110 1111 0000 X X 1111 : 500 ns;
110 1111 1111 X X 0000 : 500 ns;
%
111 0000 0000 X X 1111 : 500 ns;
111 0000 1111 X X 0000 : 500 ns;
111 1111 0000 X X 0000 : 500 ns;
111 1111 1111 X X 1111 : 500 ns;

```

Figure 8-4 WAVES test vectors for example ALU4wc.

8.1.2 Evaluation of the Behavioral Test Vectors

Example ALU4wc was first synthesized to gate level Structure1 using AutoLogic II. The resulting optimized implementation contains 59 gates shown in Figure 8-5. According to MIL-STD 883D, Structure1 of example ALU4wc contains 284 unique *SSL* gate level faults. Fault simulations using the behavioral test vectors from Figure 8-4 resulted in complete gate level fault coverage shown in Figure 8-6.

An alternate target technology was next used to synthesize gate level Structure2. The resulting optimized circuit contains 78 gates and 312 unique *SSL* faults. Fault simulations using the behavioral test vectors from Figure 8-4 again resulted in complete gate level fault coverage.

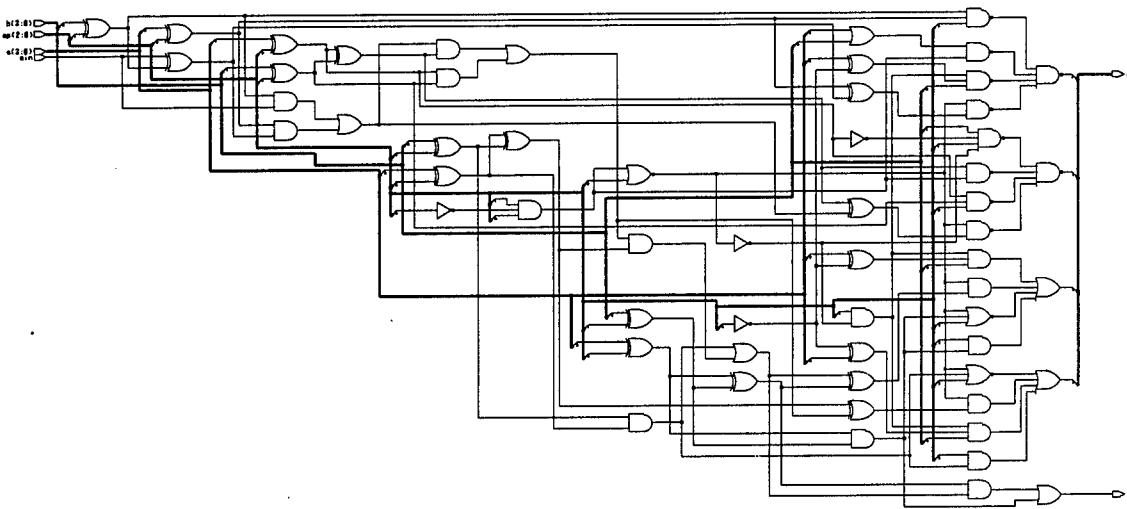


Figure 8-5 Synthesized Structure1 for example ALU4wc.

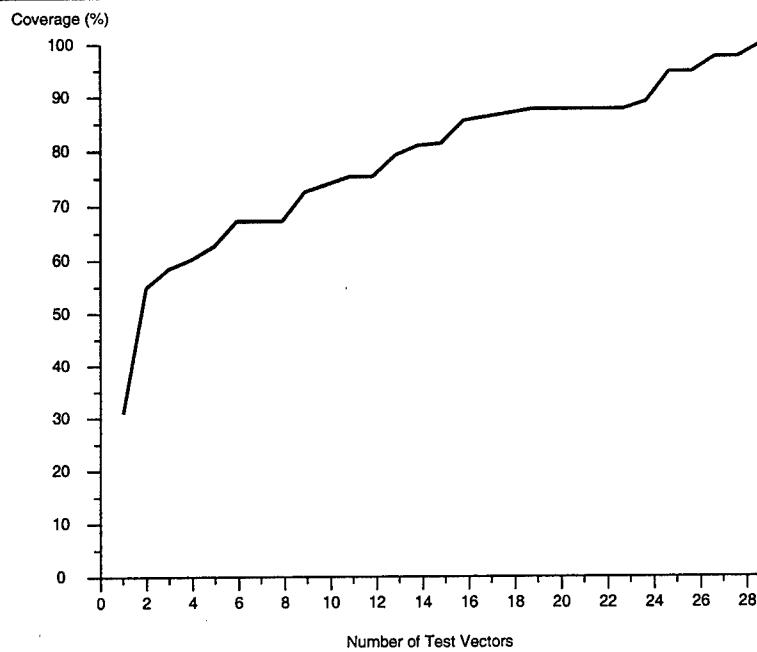


Figure 8-6 Fault coverage for Structure1 of example ALU4wc.

Lastly, an alternate synthesis tool, Leonardo, was used to map the VHDL behavioral description to a Xilinx FPGA architecture. Fault simulations using the behavioral test vectors achieved a SSL gate level fault coverage of $398/398 = 100\%$. The behavioral fault models have been applied to multiple implementations using various target architectures and synthesis tools. The range of examples demonstrates the flexibility of the approach and provides experimental validation of the effectiveness of the new fault models.

8.1.3 Expansion of the Data Path

The data path for the arithmetic logic unit was next expanded to eight bits wide to create example ALU8wc. The only difference in the resulting behavioral description is the width of the corresponding variables and signals. Since the width of the control signal *OP* remained constant, no new control faults were introduced.

The behavioral test vectors for example ALU8wc, therefore, follow directly from those derived for example ALU4wc. The only change is the expansion of the data signals *A*, *B*, and *D* to eight bits wide. The resulting test vectors are presented in Figure 8-7. Note that the WAVES file still contains only 31 test vectors.

```
% OP A B CIN COUT D : time;
000 00000000 00000000 0 0 00000000 : 500 ns;
000 11111111 11111111 1 1 11111111 : 500 ns;
000 01010101 01010101 0 0 10101010 : 500 ns;
000 10101010 10101010 1 1 01010101 : 500 ns;
%
001 11111111 00000000 0 0 00000000 : 500 ns;
001 00000000 00000000 0 0 11111111 : 500 ns;
001 10101010 01010101 0 0 10101010 : 500 ns;
001 01010101 10101010 1 1 01010101 : 500 ns;
%
010 00000000 11111111 0 0 00000000 : 500 ns;
010 00000000 00000000 0 0 11111111 : 500 ns;
010 00000000 00000000 1 1 00000000 : 500 ns;
010 11111111 11111111 1 1 00000000 : 500 ns;
%
011 00000000 00000000 0 X 00000000 : 500 ns;
011 00000000 11111111 0 X 11111111 : 500 ns;
011 11111111 00000000 0 X 11111111 : 500 ns;
%
100 00000000 11111111 0 X 00000000 : 500 ns;
100 11111111 00000000 X X 00000000 : 500 ns;
100 01010101 01010101 0 X 01010101 : 500 ns;
100 10101010 10101010 1 X 10101010 : 500 ns;
%
101 11111111 11111111 0 X 00000000 : 500 ns;
101 00000000 00000000 X X 00000000 : 500 ns;
101 10101010 01010101 0 X 01010101 : 500 ns;
101 01010101 10101010 1 X 10101010 : 500 ns;
```

Figure 8-7 WAVES test vectors for example ALU8wc.

```

110 00000000 00000000 0 X 00000000 : 500 ns;
110 00000000 11111111 0 X 11111111 : 500 ns;
110 11111111 00000000 X X 11111111 : 500 ns;
110 11111111 11111111 X X 00000000 : 500 ns;
%
111 00000000 00000000 X X 11111111 : 500 ns;
111 00000000 11111111 X X 00000000 : 500 ns;
111 11111111 00000000 X X 00000000 : 500 ns;
111 11111111 11111111 X X 11111111 : 500 ns;

```

Figure 8-7 WAVES test vectors for example ALU8wc.

Example ALU8wc was next synthesized using AutoLogic II to produce Structure1 containing 130 gates. Fault simulations using the behavioral test vectors from Figure 8-7 resulted in a SSL gate level fault coverage of $542/542 = 100\%$. Lastly, Leonardo was used to synthesize example ALU8wc to gate level Structure2. Fault simulations using the behavioral test vectors again achieved complete gate level fault coverage.

8.1.4 Summary

The new behavioral fault models, developed in this dissertation, have been applied to the comprehensive examples ALU4wc and ALU8wc. From these faults, behavioral test vectors have been derived for the 4-bit case, then readily expanded for the 8-bit example. The resulting complete *SSL* gate level fault coverage for multiple implementations is summarized in Table 8-11. Again, the range of examples demonstrates the flexibility of the approach and provides experimental validation of the effectiveness of the new fault models.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
ALU4wc	Structure1	284	31	100%
	Structure2	312	31	100%
	Structure3	398	31	100%
ALU8wc	Structure1	546	31	100%
	Structure2	758	31	100%

Table 8-11 ALU fault experiment results.

8.2 Error Correcting Circuit

The second comprehensive example is a combinational circuit capable of correcting single-bit errors in data words. The error correction capability is achieved by the use of multiple overlapping parity bits forming a Hamming code. If x is the number of information bits, then the number of parity bits, p , is equal to the smallest integer value of p that satisfies $2^P \geq x + p + 1$ [41]. Hence, four data bits (X) would require three parity bits (P) to create example HAMMING4. Likewise, eight data bits require four parity bits for example HAMMING8.

8.2.1 Example HAMMING4

The entity description for example HAMMING4 is shown in Figure 8-8. This module assumes the existence of another circuit which generates the parity bits (P) from the data bits (X). The data/parity combination X,P is then subject to corruption prior to processing by example HAMMING4. The error correcting circuit takes the input data and parity bits and performs single bit error correction to produce the output data bits (D).

```
entity hamming4 is
  port(
    X: in std_logic_vector(1 to 4);
    P: in std_logic_vector(1 to 3);
    D: out std_logic_vector(1 to 4)
  );
end hamming4;
```

Figure 8-8 Entity description for example HAMMING4.

The architecture description for example HAMMING4, shown in Figure 8-9, contains two parts. In the first section, the data bits and parity bits are combined using XOR trees to generate check bits forming a *syndrome* (S). Detection of a single-bit error produces a 1 on one or more bits of the *syndrome*. Next, the overall value of the *syndrome* bits determines which, if any, data bit needs corrected.

8.2.1.1 Faults on XOR-only Expressions

Behavioral faults on the *expressions* for the *syndrome* (S) are all on *XOR operators*. Hence, optimized test vectors can be generated based on the modified Bossen algorithm developed in Section 6.1.2.2.

```

architecture behave of hamming4 is
begin
process(X, P)
variable S : std_logic_vector(1 to 3);
begin
S(1) := X(1) XOR X(2) XOR X(4) XOR P(1);
S(2) := X(1) XOR X(3) XOR X(4) XOR P(2);
S(3) := X(2) XOR X(3) XOR X(4) XOR P(3);

D(1) <= X(1) XOR (S(1) AND S(2) AND not S(3));
D(2) <= X(2) XOR (S(1) AND not S(2) AND S(3));
D(3) <= X(3) XOR (not S(1) AND S(2) AND S(3));
D(4) <= X(4) XOR (S(1) AND S(2) AND S(3));
end process;
end behave;
```

Figure 8-9 Architecture description for example HAMMING4.

A generic 4-input *XOR*-only *expression* is shown below:

$$Z \leq A \text{ XOR } B \text{ XOR } C \text{ XOR } D$$

Applying the modified Bossen algorithm results in the labelling shown in Figure 8-10 and Figure 8-11. Note that this example represents a special case where it is possible to find identical test sequences for structures Cascade1 and Cascade2. The resulting optimized test vectors for a generic 4-input *XOR*-only *expression* are shown in Table 8-12.

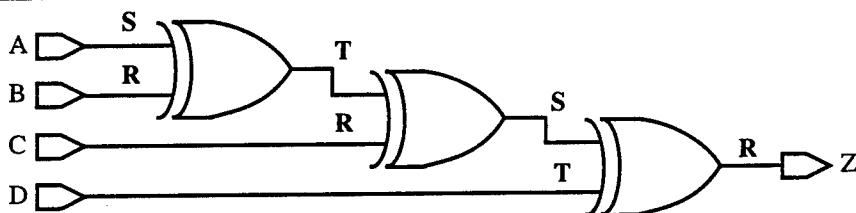


Figure 8-10 Structure Cascade1 for 4-input XOR-only expression.

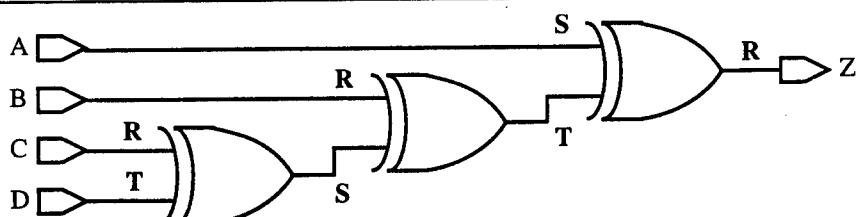


Figure 8-11 Structure Cascade2 for 4-input XOR-only expression.

Signal	Label	Sequence	Test Vector (ABCD)	Z
A	S	0011	0000	0
B	R	0110	0111	1
C	R	0110	1110	1
D	T	0101	1001	0

Table 8-12 Optimized test vectors for 4-input XOR-only expression.

Mapping the generalized case onto the first *XOR*-only *expression* relates A to $X(1)$, B to $X(2)$, C to $X(4)$, D to $P(1)$, and Z to $S(1)$. The resulting behavioral test vectors are shown in Table 8-13. Mapping the generalized case onto expressions $S(2)$ and $S(3)$ produces the test vectors shown in Table 8-14 and Table 8-15, respectively. Don't care values can be eliminated for the three *XOR*-only *expressions*, resulting in a reduced set of behavioral test vectors shown in Table 8-16.

ABCD ($X_1X_2X_4P_1$)	Z (S_1)	X ($X_1X_2X_3X_4$)	P ($P_1P_2P_3$)	S ($S_1S_2S_3$)
0000	0	00X0	0XX	0XX
0111	1	01X1	1XX	1XX
1110	1	11X1	0XX	1XX
1001	0	10X0	1XX	0XX

Table 8-13 Optimized test vectors for expression S(1).

ABCD ($X_1X_3X_4P_2$)	Z (S_2)	X ($X_1X_2X_3X_4$)	P ($P_1P_2P_3$)	S ($S_1S_2S_3$)
0000	0	0X00	X0X	X0X
0111	1	0X11	X1X	X1X
1110	1	1X11	X0X	X1X
1001	0	1X00	X1X	X0X

Table 8-14 Optimized test vectors for expression S(2).

ABCD (X ₂ X ₃ X ₄ P ₃)	Z (S ₃)	X (X ₁ X ₂ X ₃ X ₄)	P (P ₁ P ₂ P ₃)	S (S ₁ S ₂ S ₃)
0000	0	X000	XX0	XX0
0111	1	X011	XX1	XX1
1110	1	X111	XX0	XX1
1001	0	X100	XX1	XX0

Table 8-15 Optimized test vectors for expression S(3).

X ₁ X ₂ X ₃ X ₄	P ₁ P ₂ P ₃	S ₁ S ₂ S ₃
0000	000	000
0011	101	001
0100	101	000
0111	110	111
1000	110	000
1111	000	111

Table 8-16 Reduced test vector set for XOR-only expressions.

8.2.1.2 Faults on Other Logical Expressions

Behavioral faults for the remaining *expressions* are all on *logical operators*. Test vectors can be generated using the parse tree method developed in Section 6.1.1.2. A parse tree for expression D(1) is shown in Figure 8-12. The binary nodes (1,2,3) are formed by the *logical operators*.

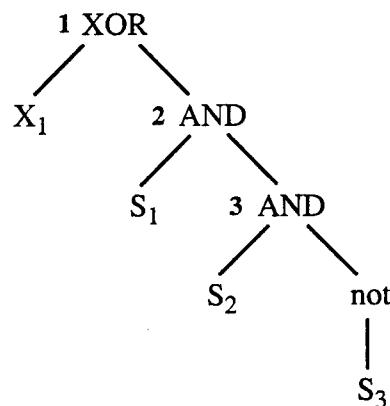


Figure 8-12 Parse tree for expression D(1).

Application of the new behavioral fault models implies four faults for the *XOR operator* and three faults for each of the *AND operators*. The resulting behavioral test vectors are presented in Table 8-17. Eliminating don't care values results in the reduced set of behavioral test vectors for expression $D(1)$ shown in Table 8-18. Application of similar parse trees to the remaining *logical expressions* produces the reduced test vectors in Table 8-19 through Table 8-21.

Node	Behavioral Test	Test Requirements	Test Vector ($X_1 S_1 S_2 S_3$)
1	00	$X_1 = 0, (S_1 \text{ AND } S_2 \text{ AND not } S_3) = 0$	0 0XX
	01	$X_1 = 0, (S_1 \text{ AND } S_2 \text{ AND not } S_3) = 1$	0 110
	10	$X_1 = 1, (S_1 \text{ AND } S_2 \text{ AND not } S_3) = 0$	1 0XX
	11	$X_1 = 1, (S_1 \text{ AND } S_2 \text{ AND not } S_3) = 1$	1 110
2	01	$S_1 = 0, (S_2 \text{ AND not } S_3) = 1, X_1 = 0$	0 010
	10	$S_1 = 1, (S_2 \text{ AND not } S_3) = 0, X_1 = 0$	0 10X
	11	$S_1 = 1, (S_2 \text{ AND not } S_3) = 1, X_1 = 0$	0 110
3	01	$S_2 = 0, \text{not } S_3 = 1, S_1 = 1, X_1 = 0$	0 100
	10	$S_2 = 1, \text{not } S_3 = 0, S_1 = 1, X_1 = 0$	0 110
	11	$S_2 = 1, \text{not } S_3 = 1, S_1 = 1, X_1 = 0$	0 111

Table 8-17 Behavioral test vectors for expression D(1).

$X_1 X_2 X_3 X_4$	$S_1 S_2 S_3$	Functional Tests (Node [Test])
0XXX	010	1[00], 2[01]
0XXX	100	2[10], 3[01]
0XXX	110	1[01], 2[11], 3[10]
0XXX	111	3[11]
1XXX	0XX	1[10]
1XXX	110	1[11]

Table 8-18 Reduced test vectors for expression D(1).

$X_1X_2X_3X_4$	$S_1S_2S_3$	Functional Tests (Node [Test])
X0XX	001	1[00], 2[01]
X0XX	100	3[10]
X0XX	101	1[01], 2[11], 3[11]
X0XX	111	2[10], 3[01]
X1XX	0XX	1[10]
X1XX	101	1[11]

Table 8-19 Reduced test vectors for expression D(2).

$X_1X_2X_3X_4$	$S_1S_2S_3$	Functional Tests (Node [Test])
XX0X	001	2[10], 3[01]
XX0X	010	3[10]
XX0X	011	1[01], 2[11], 3[11]
XX0X	111	1[00], 2[01]
XX1X	011	1[11]
XX1X	1XX	1[10]

Table 8-20 Reduced test vectors for expression D(3).

$X_1X_2X_3X_4$	$S_1S_2S_3$	Functional Tests (Node [Test])
XXX0	011	1[00], 2[01]
XXX0	101	2[10], 3[01]
XXX0	110	3[10]
XXX0	111	1[01], 2[11], 3[11]
XXX1	0XX	1[10]
XXX1	111	1[11]

Table 8-21 Reduced test vectors for expression D(4).

Eliminating don't care values by combining test vectors for logical expressions $D(1)$ through $D(4)$ produces the reduced set of behavioral test vectors shown in Table 8-22.

X ₁ X ₂ X ₃ X ₄	S ₁ S ₂ S ₃
1001	001
0101	010
1100	011
1110	011
001X	100
X010	101
X110	101
0X10	110
1X10	110
0000	111
0001	111

Table 8-22 Test vectors for logical expressions D(1) through D(4).

8.2.1.3 Final Behavioral Test Vector Set

The reduced set of behavioral test vectors for the *XOR*-only *expressions*, Table 8-16, can now be combined with the behavioral test vectors for the other *logical expressions*, Table 8-22. Unspecified don't care values are arbitrarily set to 0. The resulting test vectors were sorted and are presented in WAVES format in Figure 8-13.

```
% X P D : time;
0000 000 0000 : 500 ns;
0000 111 0001 : 500 ns;
0001 000 0000 : 500 ns;
0010 101 1010 : 500 ns;
0010 110 0110 : 500 ns;
0010 111 0010 : 500 ns;
0011 101 0011 : 500 ns;
0100 101 0100 : 500 ns;
0101 000 0101 : 500 ns;
0110 011 0010 : 500 ns;
0111 110 0110 : 500 ns;
```

Figure 8-13 WAVES test vectors for example HAMMING4.

```

1000 110 1000 : 500 ns;
1001 000 1001 : 500 ns;
1010 011 0010 : 500 ns;
1100 000 1110 : 500 ns;
1110 011 1100 : 500 ns;
1111 000 1110 : 500 ns;

```

Figure 8-13 WAVES test vectors for example HAMMING4.

8.2.2 Evaluation of the Behavioral Test Vectors

Example HAMMING4 was first synthesized to gate level Structure1 using AutoLogic II. The resulting optimized implementation contains 21 gates shown in Figure 8-14. According to MIL-STD 883D, Structure1 of example HAMMING4 contains 114 unique SSL gate level faults. Fault simulations using the behavioral test vectors from Figure 8-13 resulted in complete gate level fault coverage.

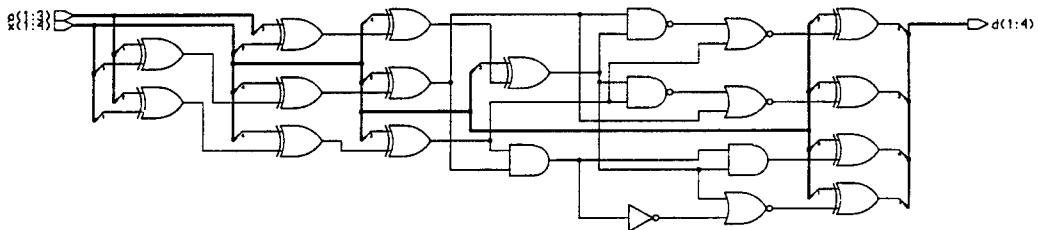


Figure 8-14 Synthesized Structure1 for example HAMMING4.

An alternate target technology was next used to synthesize gate level Structure2. The resulting optimized circuit contains 19 gates and 116 unique SSL faults. Fault simulations using the behavioral test vectors from Figure 8-13 again resulted in complete gate level fault coverage. Lastly, Leonardo was used to map the VHDL behavioral description to a Xilinx FPGA architecture. Fault simulations using the behavioral test vectors achieved a SSL gate level coverage of $226/226 = 100\%$.

8.2.3 Expansion of the Data Path

The data path for the error correcting circuit was next expanded to eight bits wide to create example HAMMING8. As previously stated, four parity bits are now required to provide single-bit error correction capability. The entity description is shown in Figure 8-15. The architecture description for example HAMMING8 is shown in Figure 8-16. Note the additional *syndrome expression* due to the 4th parity bit.

```

entity hamming8 is
    port(
        X: in std_logic_vector(1 to 8);
        P: in std_logic_vector(1 to 4);
        D: out std_logic_vector(1 to 8)
    );
end hamming8;

```

Figure 8-15 Entity description for example HAMMING8.

```

architecture behave of hamming8 is
begin
    process(X,P)
        variable S : std_logic_vector(1 to 4);
    begin
        S(1) := X(1) XOR X(2) XOR X(4) XOR X(5) XOR X(7)
                XOR P(1);
        S(2) := X(1) XOR X(3) XOR X(4) XOR X(6) XOR X(7)
                XOR P(2);
        S(3) := X(2) XOR X(3) XOR X(4) XOR X(8) XOR P(3);
        S(4) := X(5) XOR X(6) XOR X(7) XOR X(8) XOR P(4);

        D(1) <= X(1) XOR (S(1) AND S(2) AND not S(3)
                            AND not S(4));
        D(2) <= X(2) XOR (S(1) AND not S(2) AND S(3)
                            AND not S(4));
        D(3) <= X(3) XOR (not S(1) AND S(2) AND S(3)
                            AND not S(4));
        D(4) <= X(4) XOR (S(1) AND S(2) AND S(3)
                            AND not S(4));
        D(5) <= X(5) XOR (S(1) AND not S(2) AND not S(3)
                            AND S(4));
        D(6) <= X(6) XOR (not S(1) AND S(2) AND not S(3)
                            AND S(4));
        D(7) <= X(7) XOR (S(1) AND S(2) AND not S(3)
                            AND S(4));
        D(8) <= X(8) XOR (not S(1) AND not S(2) AND S(3)
                            AND S(4));
    end process;
end behave;

```

Figure 8-16 Architecture description for example HAMMING8.

None of the test vectors for example HAMMING4 can be readily expanded for use with example HAMMING8. However, deriving the behavioral test vectors follows the same process that was used in Section 8.2.1.1 and Section 8.2.1.2. Generalized Bossen

test vectors can be found for the *XOR*-only expressions and parse trees can be used for the remaining *logical expressions*. The resulting $10 + 23 = 33$ test vectors are shown in WAVES format in Figure 8-17.

```
% X      P      D      : time;
00000000 0000 00000000 : 500 ns;
01101111 0000 01111111 : 500 ns;
01110010 0010 01110000 : 500 ns;
10011101 1100 10011101 : 500 ns;
11101110 1101 01101110 : 500 ns;
11110010 1110 11110000 : 500 ns;
00100101 0011 00100100 : 500 ns;
00110110 0011 00110111 : 500 ns;
01011011 0000 01011010 : 500 ns;
01011011 0011 01011011 : 500 ns;
%
11010010 0100 11010010 : 500 ns;
10011010 0100 10011010 : 500 ns;
11011010 1111 11011011 : 500 ns;
11011011 1100 11011010 : 500 ns;
01011010 0100 01011010 : 500 ns;
11011000 0100 11011100 : 500 ns;
11011100 0001 110111000 : 500 ns;
11001010 0100 11101010 : 500 ns;
11101010 0010 11001010 : 500 ns;
11011010 1011 11011010 : 500 ns;
00100101 1000 00100101 : 500 ns;
00100101 1001 00101101 : 500 ns;
00101101 0000 00100101 : 500 ns;
00100101 1010 01100101 : 500 ns;
01100101 0000 00100101 : 500 ns;
00100100 1000 00100100 : 500 ns;
00100101 1100 10100101 : 500 ns;
10100101 0000 00100101 : 500 ns;
00100001 1000 00100011 : 500 ns;
00100011 0101 00100001 : 500 ns;
00000101 1000 00010101 : 500 ns;
00010101 0110 00000101 : 500 ns;
00100101 1111 00100101 : 500 ns;
```

Figure 8-17 WAVES test vectors for example HAMMING8.

Example HAMMING8 was next synthesized using AutoLogic II to produce Structure1 containing 36 gates. Fault simulations using the behavioral test vectors from Figure 8-17

resulted in a *SSL* gate level fault coverage of $226/226 = 100\%$. Lastly, Leonardo was used to synthesize example HAMMING8 to gate level Structure2. Fault simulations using the behavioral test vectors again achieved complete gate level fault coverage.

8.2.4 Summary

The new behavioral fault models, developed in this dissertation, have been applied to the comprehensive examples HAMMING4 and HAMMING8. From these faults, behavioral test vectors were derived for both the 4-bit and 8-bit cases. The resulting complete *SSL* gate level fault coverage for multiple implementations is summarized in Table 8-23.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
HAMMING4	Structure1	114	17	100%
	Structure2	116	17	100%
	Structure3	226	17	100%
HAMMING8	Structure1	226	33	100%
	Structure2	446	33	100%

Table 8-23 HAMMING fault experiments.

8.3 Conclusions

Two comprehensive examples were chosen to demonstrate the gate level fault coverage of the new behavioral fault models. The ALU involved the interaction of control faults with both *arithmetic* and *logical operator* faults. The single error correcting circuit, HAMMING, used both *XOR*-only and mixed *logical operator expressions*. Application of the new fault models to the comprehensive examples resulted in sets of test vectors necessary to detect the behavioral faults. These test vectors were then applied to synthesized gate level implementations of the behavioral descriptions. The resulting complete *SSL* gate level fault coverage provides experimental validation of the effectiveness of the behavioral fault models.

Chapter 9

Conclusions and Future Work

This chapter summarizes the research contributions of this dissertation and outlines directions for future work. Some concluding remarks are also provided.

9.1 Research Contributions

The main contributions of this dissertation include improved behavioral fault models as well as the techniques for generalizing the effects of low level faults and abstracting them into the behavioral domain. The new fault models are more closely linked to underlying hardware faults than those developed by previous research. Test vectors based on these new behavioral fault models achieve complete *SSL* gate level fault coverage over a broad range of implementations.

9.1.1 Generalized Functional Faults

A functional analysis technique has been developed for generalizing the effects of industry standard *single-stuck-line* (*SSL*) faults on gate level circuits. The key is determining sets of functional faults which are not tied to a specific realization. What is desired is a general set of faults which provide coverage for functional faults from multiple implementations.

For regular structures, such as cellular logic arrays and parity trees, complete functional testing is achieved by exhaustive testing of each functional building block. For other functions, faults are generalized from *sum-of-products* (*SOP*) and *product-of-sums* (*POS*) implementations to obtain a set of functional faults not tied to a specific realization. As was the case with *physically-induced* faults [29], a unique fault produced by a particular realization can be readily added to the set of functional faults.

This dissertation has used the *SSL* fault model as the basis for its higher level fault models. The generalization and abstraction techniques developed here are not dependent on this choice of a low level fault model. Other low level fault models which complement or improve on the *SSL* fault model can also be readily applied.

9.1.2 New Behavioral Fault Models

New behavioral fault models have been developed, which are more closely linked to underlying hardware faults than previous fault models. The effects of the generalized sets of functional faults are abstracted into the behavioral domain by establishing a relationship between the higher level language construct and the lower level faults it should encompass.

The fault modeling technique used throughout this dissertation is that of external corruption of the original VHDL constructs, rather than replacement/mutation of operators. Where possible, specific faults have been defined, such as the *Clause-CORRUPT* control faults. When a direct mapping of functional faults cannot be made to produce a simple behavioral fault model, an error vector approach has been applied. The functional test vectors are mapped into error vectors which then corrupt the results of the VHDL operation for the appropriate input combinations.

While the new fault models are definitely more complex than previous ones, this is because they more accurately reflect the underlying complexity of the hardware faults which they attempt to model. The increased complexity of the fault models eliminates the need to supplement behavioral test vector sets via heuristics in order to improve gate level fault coverage.

9.1.3 Gate Level Fault Coverage of Behavioral Test Vectors

Application of the behavioral fault models to examples throughout this dissertation resulted in sets of test vectors necessary to detect the behavioral faults. Fault experiments were then performed using the behavioral test vectors and synthesized gate level implementations. Multiple synthesis tools and target architectures were employed to create a broad range of realizations of the behavioral descriptions. Resulting gate level fault coverage was evaluated to illustrate the effectiveness of the behavioral fault models and is summarized in Appendix B.

Two comprehensive examples were chosen to demonstrate the gate level fault coverage of the new behavioral fault models. The ALU involved the interaction of control faults with both *arithmetic* and *logical operator* faults. The single error correcting circuit, HAMMING, used both *XOR*-only and mixed *logical operator expressions*. Application of

the new fault models to the comprehensive examples resulted in sets of test vectors necessary to detect the behavioral faults. These test vectors were then applied to synthesized gate level implementations of the behavioral descriptions. The resulting complete SSL gate level fault coverage provides experimental validation of the effectiveness of the behavioral fault models.

9.1.4 Behavioral Test Generation

The base fault model for *arithmetic operators* is derived from the *ripple carry* connection of 2- and 3-input functional building blocks. Behavioral test generation rules, presented in this dissertation, demonstrate that only eight test vectors are required for complete gate level fault coverage, regardless of the size of the *operands*. For *logical operators*, behavioral test generation rules were developed for the special case of *XOR*-only *expressions*. A generalized Bossen algorithm is presented that allows for optimization of test sequences while allowing for multiple possible realizations.

Most fault-oriented techniques use some form of a three step approach to the test generation process. First, a fault must be *activated* at the desired location in the circuit model. Then, the effect of the fault must be *propagated* to a point where it can be observed and, hence, detected. Finally, the inputs of the model must be determined to *justify* the desired signal values throughout the circuit. Variations of these techniques attempt to utilize the information available in higher level models to more efficiently accomplish the computationally intensive tasks of fault *propagation* and *justification*.

The behavioral fault models developed in this dissertation only affect the *activation* step of the test generation process. Hence, the high level algorithms developed to handle the *propagation* and *justification* steps still remain valid. Integration of the new behavioral fault models with a behavioral test generation algorithm such as the B-algorithm [21][22] can be of mutual benefit. Such advanced test generation algorithms already address problems such as reconvergent fanout, while use of more complex fault models can eliminate the need to supplement test vector sets via heuristics.

9.1.5 Behavioral Fault Simulation

The new behavioral fault models developed in this dissertation can now be integrated with fault injection techniques such as those developed by DeLong et al. [23][24] to allow

fault simulation at higher levels of abstraction. Because the fault models were developed using an external corruption or error vector approach, modifications to the original VHDL behavioral description need only be made once. Individual faults in the compiled VHDL fault simulation model can then be activated by external manipulation of the corrupting signals/error vectors.

The ability to fault simulate VHDL behavioral models rather than more detailed gate level ones will allow better management of ever increasing design complexity. Working with behavioral fault models will also allow fault simulation to be performed earlier in the design scheme, without details of the gate level implementation. In fact, depending on the source of the component, a gate level description may never be available. Thus, these new behavioral fault models facilitate better integration of fault simulation into the overall design process.

9.2 Future Work

The models and techniques presented in this dissertation represent another important step in the development of a design methodology for performing fault simulation throughout the design process. The following sections present a brief description of directions for future research.

9.2.1 Expansion of Behavioral Fault Models

The fault models developed in this dissertation address combinational logic circuits based on the *IEEE Draft Standard for VHDL Register Transfer Level Synthesis* [36]. These behavioral models need to be expanded to include sequential components. The draft standard includes a set of representative design examples whose intent is to specify certain prevalent modeling styles resulting in basic hardware elements like flip-flops, latches, etc. The specification of processes and resulting inferred logic are detailed in Appendix D.

This dissertation has used the *SSL* fault model as the basis for its higher level fault models. The generalization and abstraction techniques developed here are not dependent on this choice of a low level fault model. Other low level fault models which complement or improve on the *SSL* fault model can also be readily applied. Additional fault effects may be abstracted into the behavioral domain, thus improving the overall behavioral fault

models. Fault coverage metrics would have to be adjusted accordingly to effectively represent the low level fault coverage of the improved behavioral fault models.

9.2.2 Tool Development

As previously stated, the behavioral fault models developed in this dissertation can now be integrated into higher level test generation algorithms. The resulting behavioral test generation tool would allow designers to develop test vector sets based on VHDL behavioral descriptions. These behavioral test vector sets could then be used to fault simulate a component at the gate level or even used to test components for which a gate level description is not available.

The new behavioral fault models developed in this dissertation can also be combined with fault injection techniques to allow fault simulation at higher levels of abstraction. A VHDL behavioral fault simulation tool would allow fault experiments to be performed earlier in the design scheme, without details of the gate level implementation. Both behavioral test generation and behavioral fault simulation tools will aid in complexity management and better integrate fault simulation into the overall design process.

9.2.3 Higher Levels of Abstraction

The VHDL descriptions and subsequent fault models in this dissertation cross from the structural into the behavioral domain and move up the design hierarchy from the gate to the register level as defined in Table 1-1. A logical extension to this work is the continuation to higher levels of abstraction such as the chip or system level. The further migration of fault models would clearly support the ultimate goal of developing a design methodology for performing fault simulation throughout the design process.

The design tools used in this dissertation involved synthesis of behavioral data flow descriptions into structural gate level circuits. Moving higher up the design hierarchy next involves algorithmic synthesis tools which translate chip level algorithms into data flow descriptions. Understanding this synthesis process is key to further abstraction of the behavioral fault models.

Ultimately the level of abstraction is reached where the VHDL description is that of a combined hardware/software system. Fault models need to be developed at this system

level so fault simulation and test generation can be integrated into the hardware/software codesign process.

9.3 Concluding Remarks

To cope with the ever increasing complexity of digital circuits, engineers can now work at higher levels of abstraction by taking advantage of computer aided design packages and hardware description languages. Sophisticated synthesis tools provide a design environment which allows the use of higher level VHDL behavioral models. The details of the gate level implementation are safely hidden, shielding the designer from additional complexity. The fault models and abstraction techniques developed in this dissertation represent another important step in integrating fault simulation and testing into such a VHDL synthesis environment.

Expansion of these new behavioral fault models and development of associated computer-aided tools, will allow better management of design complexity. Fault simulation and testing of digital circuits can be moved away from the traditional gate level to join other design aspects at higher levels of abstraction. The end result will be a design methodology which includes performing fault simulation throughout the entire design process.

References

- [1] Abraham, J.A. and V.K. Agarwal, "Test Generation for Digital Systems," *Fault-Tolerant Computing: Theory and Techniques*, D. K. Pradhan, ed., Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [2] Abraham, J.A. and K. Fuchs, "Fault and Error Models for VLSI," *Proceedings of the IEEE*, Vol. 74, No. 5, May 1986, pp. 639-654.
- [3] Abramovici, M., M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, Woodland Hills, CA, 1990.
- [4] Al Hayek, G. and C. Robach, "On the Adequacy of Deriving Hardware Test Data from the Behavioral Specification," *Proceedings EUROMICRO 96, 22nd Euromicro Conference*, September 1996, pp. 337-342.
- [5] Al Hayek, G. and C. Robach, *A Mutation-based Test for VHDL Descriptions*, technical report, LSR-IMAG, 1995.
- [6] Armstrong, J.R., "Chip Level Modeling of LSI Devices," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-3, No. 4, October 1984, pp. 288-297.
- [7] Armstrong, J.R., "Chip Level Modeling with HDL's," *IEEE Design & Test of Computers*, Vol. 5, No. 1, February 1988, pp. 8-18.
- [8] Armstrong, J.R., *Chip Level Modeling with VHDL*, Prentice Hall, Englewood Cliffs, NJ, August 1989.
- [9] Armstrong, J.R., F.S. Lam, and P.C. Ward, "Test Generation and Fault Simulation for Behavioral Models," *Performance and Fault Modeling with VHDL*, J.M. Schoen, ed., Prentice Hall, Englewood Cliffs, NJ, 1992, pp. 240-303.
- [10] Armstrong, J.R. and F.G. Gray, *Structured Logic Design with VHDL*, Prentice Hall, Englewood Cliffs, NJ, 1993.
- [11] Ashenden, P.J., *The VHDL Cookbook*, technical report, Department of Computer Science, University of Adelaide, South Australia, 1990.

- [12] Banerjee, P., *A Model for Simulating Physical Failures in MOS VLSI Circuits*, Technical Report CSG-13, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1985.
- [13] Barclay, D.S. and J.R. Armstrong, "A Heuristic Chip-level Test Generation Algorithm," *Proceedings 23rd Design Automation Conference*, June 1986, pp. 257-262.
- [14] Barclay, D.S., *An Automatic Test Generation Method for Chip-Level Circuit Descriptions*, master's thesis, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, VA, January 1987.
- [15] Baweja, G., *Gate Level Coverage of A Behavioral Test Generator*, master's thesis, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, VA, March 1993.
- [16] Bossen, D.C., D.L. Ostapko, and A.M. Patel, "Optimum Test Patterns for Parity Networks," *Proceedings AFIPS Fall Joint Computer Conference*, Vol. 37, November 1970, pp. 33-38.
- [17] Brahme, D. and J.A. Abraham, "Functional Testing of Microprocessors," *IEEE Transactions on Computers*, Vol C-33, June 1984, pp. 475-485.
- [18] Chakraborty, T. and S. Ghosh, "On Behavior Fault Modeling for Combinational Digital Designs," *Proceedings International Test Conference*, September 1988, pp. 593-600.
- [19] Chao, C.H. and F.G. Gray, "Micro-Operation Perturbations in Chip Level Fault Modeling," *Proceedings 25th Design Automation Conference*, 1988, pp. 579-582.
- [20] Chen, C.-I.H. and S. Perumal, "Analysis of the Gap between Behavioral and Gate-Level Fault Simulation," *Proceedings Sixth Annual IEEE International ASIC Conference and Exhibit*, September 1993, pp. 144-147.
- [21] Cho, C.H., *A Formal Model for Behavioral Test Generation*, doctoral dissertation, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, VA, February 1994.

- [22] Cho, C.H. and J.R. Armstrong, "B-algorithm: A Behavioral Test Generation Algorithm," *Proceedings International Test Conference*, 1994, pp. 968-979.
- [23] DeLong, T.A., B.W. Johnson, and J.A. Profeta, III, "A Fault Injection Technique for VHDL Behavioral-Level Models," *IEEE Design & Test of Computers*, Vol. 13, No. 4, Winter 1996, pp. 24-33.
- [24] DeLong, T.A., D.T. Smith, B.W. Johnson, and J.P. Hanna, "Simulator Independent Fault Simulation Using WAVES," *Proceedings Fall '96 VIUF Conference*, October 1996, pp. 129-138.
- [25] ESIP, *Level-0 VHDL Synthesis Syntax and Semantics*, technical report, Microelectronics Group, University of Cantabria, December 1995.
- [26] Ghosh, S. "Behavior-Level Fault Simulation," *IEEE Design & Test of Computers*, June 1988, pp. 31-42.
- [27] Ghosh, S. and T.J. Chakraborty, "On Behavior Fault Modeling for Digital Designs," *Journal of Electronic Testing*, Vol. 2, Kluwer Academic Publishers, 1991.
- [28] Gupta, A.K. and J.R. Armstrong, "Functional Fault Modeling and Simulation for VLSI Devices," *Proceedings 22nd Design Automation Conference*, 1985, pp. 720-726.
- [29] Hansen, M.C. and J.P. Hayes, "High-Level Test Generation using Physically-Induced Faults," *Proceedings VLSI Test Symposium*, May 1995, pp. 20-28.
- [30] Hayes, J.P., "On the Realization of Boolean Functions Requiring a Minimal or Near-minimal Number of Tests," *IEEE Transactions on Computers*, Vol. C-20, December 1971, pp. 1506-1513.
- [31] Hayes, J.P., "Modeling Faults in Digital Logic Circuits," *Rational Fault Analysis*, R. Saeks and S. R. Liberty, eds., Marcel Dekker, New York, NY, 1977, pp. 78-95.
- [32] Hayes, J.P., "A Fault Simulation Methodology for VLSI," *Proceedings 19th Design Automation Conference*, June 1982, pp. 393-399.

- [33] Hayes, J.P., "Fault Modeling for Digital MOS Integrated Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-3, No. 3, July 1984, pp. 200-207.
- [34] Hayes, J.P., "Fault Modeling," *IEEE Design & Test*, Vol. 2, No. 2, April 1985, pp. 37-44.
- [35] Herzog, J.H., *Design and Organization of Computing Structures*, Franklin, Beedle & Associates, Wilsonville, OR, 1996.
- [36] *IEEE P1076.6/D1.12, Draft Standard for VHDL Register Transfer Level Synthesis*, VHDL Synthesis Interoperability Working Group, IEEE, Piscataway, NJ, March 1998.
- [37] *IEEE Std 1029.1-1991, IEEE Standard for Waveform and Vector Exchange (WAVES)*, IEEE, New York, NY, 1991.
- [38] *IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual*, IEEE, New York, NY, 1988.
- [39] Jenn, E., J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," *Proceedings 24th International Symposium on Fault-Tolerant Computing*, June 1994, pp. 66-75.
- [40] Johnson, B.W., D.T. Smith, and T.A. DeLong, *A Survey of Fault Simulation, Fault Grading, and Test Pattern Generation Techniques with an Emphasis on the Feasibility of VHDL Based Fault Simulation*, University of Virginia, Center for Semicustom Integrated Systems, Department of Electrical Engineering, Charlottesville, VA, Report to Rome Laboratory/PKRZ, May 1996.
- [41] Johnson, E.L. and M.A. Karim, *Digital Design, A Pragmatic Approach*, PWS Engineering, Boston, MA, 1987.
- [42] Johnson, W., "Behavioral-Level Test Development," *Proceedings of the 16th Annual Design Automation Conference*, 1979, pp. 171-179.
- [43] Kalia, A., *Cadence VHDL Synthesizable Subset*, technical report, VHDL Synthesis Interoperability Working Group, August 1996.

- [44] Kautz, W.H., "Testing for Faults in Cellular Logic Arrays," *Proceedings 8th Annual Symposium Switching, Automata Theory*, 1967, pp. 161-174.
- [45] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill, New York, NY, 1978.
- [46] Lam, F.S., *Test Generation for Behavioral Models with Reconvergent Fanouts and Feedback*, master's thesis, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, VA, September 1989.
- [47] *Leonardo Synthesis and Technology Guide*, Release 4.0, Exemplar Logic, Inc., Alameda, CA, 1996.
- [48] Levendel, Y.H. and P.R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages," *IEEE Transactions on Computers*, Vol. C-31, pp. 577-589, July 1982.
- [49] Lin, T., *Functional Test Generation of Digital LSI/VLSI Systems Using Machine Symbolic Execution Technique*, doctoral dissertation, SUNY-Binghamton, Computer Science Department, Binghamton, NY, 1985.
- [50] Lin, T. and S.Y.H. Su, "The S-Algorithm: A Promising Solution for Systematic Functional Test Generation," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-4, July 1985, pp. 250-263.
- [51] McCluskey, E.J., *Logic Design Principles with Emphasis on Testable Semi-custom Circuits*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [52] *MIL-STD-883D, Test Methods and Procedures for Microelectronics*, Method 5012.1, November 1991.
- [53] Mourad, S. and E.J. McCluskey, "Testability of Parity Checkers," *IEEE Transactions on Industrial Electronics*, Vol. 36, No. 2, May 1989, pp. 254-262.
- [54] Noh, T.H., C.-I.H. Chen, and S.M. Chung, "Behavioral Fault Simulation and ATPG System for VHDL," *Proceedings Seventh Annual IEEE International ASIC Conference and Exhibit*, 1994, pp. 412-416.

- [55] Norrod, F.E., *The E-Algorithm, an Automatic Test Generation Algorithm for Hardware Description Languages*, master's thesis, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, VA, February 1988.
- [56] Norrod, F.E., "An Automatic Test Generation Algorithm for Hardware Description Languages," *Proceedings 26th ACM/IEEE Design Automation Conference*, June 1989, pp. 429-434.
- [57] O'Neill, M.D., An Improved Chip-Level Test Generation Algorithm, master's thesis, Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, VA, January 1988.
- [58] O'Neill, M.D., D.D. Jani, C.H. Cho, and J.R. Armstrong, "BTG: A Behavioral Test Generator," *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages and their Applications*, June 1989, pp. 347-361.
- [59] Pla, V., J.-F. Santucci, and N. Giambiasi, "On the Modeling and Testing of VHDL Behavioral Descriptions of Sequential Circuits," *Proceedings European Design Automation Conference*, September 1993, pp. 440-445.
- [60] Riesgo, T. and J. Uceda, "A Fault Model for VHDL Descriptions at the Register Transfer Level," *European Design Automation Conference*, September 1996, pp. 462-467.
- [61] Santucci, J.F., A.L. Courbis, and N. Giambiasi, "Behavioral Testing of Digital Circuits," *Journal of Microelectronic Systems Integration*, Vol. 1, No. 1, 1993, pp. 55-77.
- [62] Shen, J.P., W. Maly, and F.J. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits," *IEEE Design & Test*, Vol. 2, December 1985, pp. 13-26.
- [63] Sridhar, T. and J.P. Hayes, "A Functional Approach to Testing Bit-Sliced Microprocessors," *IEEE Transactions on Computers*, Vol. C-30, No. 8, August 1981, pp. 563-571.

- [64] *Standard for Synthesizing from VHDL Language at the Register Transfer Level*, Synopsys, Mountain View, CA, October 1996.
- [65] Thatte, S.M. and J.A. Abraham, "A Methodology for Functional Level Testing of Microprocessors," *Proceedings 8th International Conference on Fault-Tolerant Computing*, June 1978, pp. 90-95.
- [66] Thatte, S.M. and J.A. Abraham, "Test Generation for Microprocessors," *IEEE Transactions on Computers*, Vol. C-29, No. 6, June 1980, pp. 429-441.
- [67] *The TTL Data Book*, 2nd Ed., Texas Instruments, Inc., Dallas, TX, 1976.
- [68] *VHDL Style Guide for AutoLogic II*, Mentor Graphics Corporation, Wilsonville, OR, 1995.
- [69] Ward, P.C. and J.R. Armstrong, "Behavioral Fault Simulation in VHDL," *Proceedings 27th Design Automation Conference*, June 1990, pp. 587-593.
- [70] Yount, C.R., *The Automatic Generation of Instruction-Level Error Manifestations of Hardware Faults: A New Fault-Injection Model*, doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA, May 1993.
- [71] Yount, C.R. and D.P. Siewiorek, "A Methodology for the Rapid Injection of Transient Hardware Errors," *IEEE Transactions on Computers*, Vol. 45, No. 8, August 1996, pp. 881-891.

Appendix A

Additional Examples

In order to more fully examine the effectiveness of the new behavioral fault models, additional examples are provided here. Examples have been chosen to represent a broad range of design possibilities. Multiple synthesis options are employed to ensure the examples are as general as possible.

A.1 Array Indexing

As shown in Appendix E, indexing an array such as a bit_vector also implies a multiplexer architecture. Consider the VHDL behavioral description for example ARRAY4 shown in Figure A-1.

```
entity ARRAY4 is
    port(Y: in BIT_VECTOR(3 downto 0);
         I: in INTEGER range 3 downto 0;
         Z: out BIT);
end ARRAY4;

architecture BEHAVE of ARRAY4 is
begin
    process(Y,I)
    begin
        Z <= Y(I);
    end process;
end BEHAVE;
```

Figure A-1 Behavioral description for example ARRAY4.

The assignment statement `Z <= Y(I)` can be equivalently written as a *case* statement, as shown in Figure A-2.

```
case I is
    when 0 => Z <= Y(0);
    when 1 => Z <= Y(1);
    when 2 => Z <= Y(2);
    when 3 => Z <= Y(3);
end case;
```

Figure A-2 Equivalent case statement for example ARRAY4.

This selection activity implies the applicability of the control fault model developed in Chapter 3. According to the model, each *selection* (clause or index value) can be affected by two different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. According to a binary encoding for I , specified by the synthesis subset, Figure A-3 shows the resulting logical adjacencies for this example.

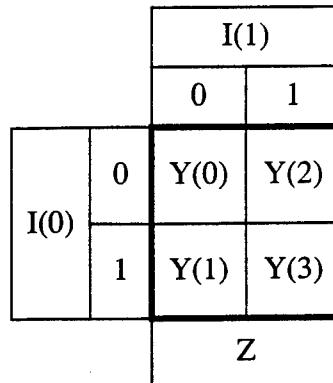


Figure A-3 Logical adjacencies among clauses.

Thus, applying the control fault model to example ARRAY4 in Figure A-1, results in the behavioral faults shown in Table A-1.

Behavioral Fault	Corrupted Selection	Test Vector (I Y)
Y(0)-CORRUPT (by Y(1))(OR)	$Z \leq Y(0) \text{ OR } Y(1)$	0 XX10
Y(0)-CORRUPT (by Y(1))(AND)	$Z \leq Y(0) \text{ AND } Y(1)$	0 XX01
Y(0)-CORRUPT (by Y(2))(OR)	$Z \leq Y(0) \text{ OR } Y(2)$	0 X1X0
Y(0)-CORRUPT (by Y(2))(AND)	$Z \leq Y(0) \text{ AND } Y(2)$	0 X0X1
Y(1)-CORRUPT (by Y(0))(OR)	$Z \leq Y(1) \text{ OR } Y(0)$	1 XX01
Y(1)-CORRUPT (by Y(0))(AND)	$Z \leq Y(1) \text{ AND } Y(0)$	1 XX10
Y(1)-CORRUPT (by Y(3))(OR)	$Z \leq Y(1) \text{ OR } Y(3)$	1 1X0X
Y(1)-CORRUPT (by Y(3))(AND)	$Z \leq Y(1) \text{ AND } Y(3)$	1 0X1X
Y(2)-CORRUPT (by Y(0))(OR)	$Z \leq Y(2) \text{ OR } Y(0)$	2 X0X1
Y(2)-CORRUPT (by Y(0))(AND)	$Z \leq Y(2) \text{ AND } Y(0)$	2 X1X0
Y(2)-CORRUPT (by Y(3))(OR)	$Z \leq Y(2) \text{ OR } Y(3)$	2 10XX
Y(2)-CORRUPT (by Y(3))(AND)	$Z \leq Y(2) \text{ AND } Y(3)$	2 01XX

Table A-1 Behavioral faults for example ARRAY4.

Behavioral Fault	Corrupted Selection	Test Vector (I Y)
Y(3)-CORRUPT (by Y(1))(OR)	$Z \leq Y(3) \text{ OR } Y(1)$	3 0X1X
Y(3)-CORRUPT (by Y(1))(AND)	$Z \leq Y(3) \text{ AND } Y(1)$	3 1X0X
Y(3)-CORRUPT (by Y(2))(OR)	$Z \leq Y(3) \text{ OR } Y(2)$	3 01XX
Y(3)-CORRUPT (by Y(2))(AND)	$Z \leq Y(3) \text{ AND } Y(2)$	3 10XX

Table A-1 Behavioral faults for example ARRAY4.

Combining don't care values produces the behavioral test vectors shown in WAVES format in Figure A-4.

```
% Y  I  Z : time;
X110 00 0 : 500 ns;
X001 00 1 : 500 ns;
1X01 01 0 : 500 ns;
0X10 01 1 : 500 ns;
10X1 10 0 : 500 ns;
01X0 10 1 : 500 ns;
011X 11 0 : 500 ns;
100X 11 1 : 500 ns;
```

Figure A-4 WAVES test vectors for example ARRAY4.

Example ARRAY4 was synthesized to the gate level circuit shown in Figure A-5. Fault simulations were performed using the behavioral test vectors derived from the control fault model resulting in a SSL gate level fault coverage of $44/44 = 100\%$.

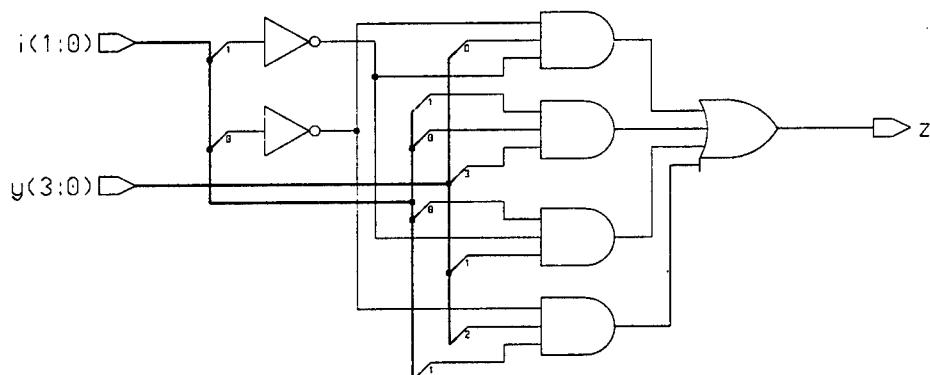


Figure A-5 Synthesized circuit for example ARRAY4.

A.2 Generalization of the Control Fault Model

Previous examples have used explicit values of control signals to determine selection in *if-then-else* and *case* statements. When working in the behavioral domain, the specific combination of control inputs may not matter, hence, an enumerated type may be used. The assignment of control signal values to the elements of the enumerated type can be left to the design tools and, thus, delayed to later in the design cycle. The new control fault model can be easily generalized to allow for use of enumerated types.

Example CASE2, in Figure A-6, uses an enumerated type to control the selection of assignments for a *case* statement.

```

case SEL is
  when J =>
    Z <= Y0;
  when K =>
    Z <= Y1;
  when L =>
    Z <= Y2;
  when M =>
    Z <= Y3;
end case;

```

Figure A-6 Behavioral description for example CASE2.

Due to the independence of the input signals in this example, the synthesis tool produces the standard implementation of the *case* statement as a 4-to-1 multiplexer. Each of the four inputs (Y_3, Y_2, Y_1, Y_0) can be assigned to any of the four multiplexer inputs (A, B, C, D), depending on the designation of control bits for the enumerated type *SEL*. There are, therefore, $4! = 24$ possible permutations of the eventual synthesized circuit.

The control fault model can still be applied to the generalized *case* statement in example CASE2, without consideration of the eventual designation of the control bits for the enumerated type *SEL*. According to the fault model, each clause of a *case* statement is affected by two different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. The only difference with this example is the determination of logical adjacencies between clauses.

Since no assignment of control values has yet been made for the enumerated type *SEL*, actual determination of adjacencies among clauses cannot be made. It must, therefore, be

assumed that each clause has the possibility of being adjacent to, and hence corrupted by, any other clause. This assumption implies that each of the four clauses in this example is affected by three *WHEN-CORRUPT (OR)* faults and three *WHEN-CORRUPT (AND)* faults. For example, the (*OR*) corruption of the *when K* clause results in three behavioral faults: *WHEN-K-CORRUPT (by J)(OR)*, *WHEN-K-CORRUPT (by L)(OR)*, and *WHEN-K-CORRUPT (by M)(OR)*. Application of the control fault model to the generalized *case* statement from example CASE2, therefore, results in a total of 24 behavioral faults.

Though the application of the control fault model to the generalized *case* statement has resulted in the definition of eight additional behavioral faults, recall also the concept of *compatible fault sets*. The group of faults that corrupt a single channel can form a *compatible fault set* and their test vectors can, therefore, be combined. For example, the three *WHEN-CORRUPT (OR)* faults that affect clause *K* lead to the derivation of the following test vectors (*SEL Y3 Y2 Y1 Y0*): *KXX01*, *KX10X*, and *K1X0X*. Though in the actual circuit, only two of the other clauses will be adjacent to clause *K*, the three test vectors are still compatible, forming the test vector *K1101*. Comparison of this test vector with the corresponding one derived for the *Z <= Y1* clause for example IF2, (*SEL Y3 Y2 Y1 Y0*) *011X01*, shows that the only difference is the elimination of an additional don't care. Alternate assignment of control signals for *K* simply alter the adjacencies between clauses and result in the elimination of a different don't care. A list of *WHEN-CORRUPT (OR)* behavioral faults and their corresponding test vectors is given in Table A-2. A similar list can be easily derived for the 12 *WHEN-CORRUPT (AND)* faults.

Behavioral Fault	Test Vector (<i>SEL Y3 Y2 Y1 Y0</i>)
WHEN-J-CORRUPT (by K)(OR)	J 1110
WHEN-J-CORRUPT (by L)(OR)	
WHEN-J-CORRUPT (by M)(OR)	
WHEN-K-CORRUPT (by J)(OR)	K 1101
WHEN-K-CORRUPT (by L)(OR)	
WHEN-K-CORRUPT (by M)(OR)	

Table A-2 Behavioral faults and corresponding test vectors for example CASE2.

Behavioral Fault	Test Vector (SEL Y3 Y2 Y1 Y0)
WHEN-L-CORRUPT (by J)(OR)	L 1011
WHEN-L-CORRUPT (by K)(OR)	
WHEN-L-CORRUPT (by M)(OR)	
WHEN-M-CORRUPT (by J)(OR)	M 0111
WHEN-M-CORRUPT (by K)(OR)	
WHEN-M-CORRUPT (by L)(OR)	

Table A-2 Behavioral faults and corresponding test vectors for example CASE2.

The control fault model has now been generalized to handle enumerated types and the delay of assignment of control signal values. The only change to the model was the inclusion of additional *Clause-CORRUPT* faults due to the assumption that, in the general case, each clause may be adjacent to any other clause. The additional behavioral faults did not, necessarily, result in any additional test vectors due to the concept of *compatible fault sets*.

A.3 Signed Comparison

Example GREATER3 is presented here to demonstrate the application of the behavioral fault models for relational operators to signed as well as unsigned comparisons. As can be seen in Figure A-7, example GREATER3 compares two integers with ranges of -4 to +3. These control signals will be synthesized as 3-bit 2's complement numbers.

```

entity GREATER3 is
  port(A,B: in INTEGER range -4 to +3; GT: out BIT);
end GREATER3;
architecture BEHAVE of GREATER3 is
begin
  process(A,B)
  begin
    if A > B then
      GT <= '1';
    else
      GT <= '0';
    end if;
  end process;
end BEHAVE;

```

Figure A-7 Behavioral description for example GREATER3.

Applying the behavioral faults for unsigned *relational operators* from Chapter 4 gives the fault classifications for a signed *GT* function shown in Figure A-8.

		A							
		-4	-3	-2	-1	0	1	2	3
B		-4	I	III					
		-3		I	III				
		-2		II	I	III			
		-1			I	III			
		0			II	I	III		
		1					I	III	
		2					II	I	III
		3							I

Figure A-8 Fault classes for 3-bit signed GT function.

Encoding the test vectors as 3-bit 2's complement numbers results in the WAVES file shown in Figure A-9.

```
% A B  GT : time;
% Class I
100 100 0 : 500 ns;
101 101 0 : 500 ns;
110 110 0 : 500 ns;
111 111 0 : 500 ns;
000 000 0 : 500 ns;
001 001 0 : 500 ns;
010 010 0 : 500 ns;
011 011 0 : 500 ns;

% Class II
101 110 0 : 500 ns;
111 000 0 : 500 ns;
001 010 0 : 500 ns;
```

Figure A-9 WAVES test vectors for example GREATER3.

```
% Class III
101 100 1 : 500 ns;
110 101 1 : 500 ns;
111 110 1 : 500 ns;
000 111 1 : 500 ns;
001 000 1 : 500 ns;
010 001 1 : 500 ns;
011 010 1 : 500 ns;
```

Figure A-9 WAVES test vectors for example GREATER3.

Example GREATER3 was synthesized and optimized to gate level Structure1, shown in Figure A-10, using AutoLogic II. In order to add even more diversity to the problem, an alternate design library from that used for example COMPARE was chosen.

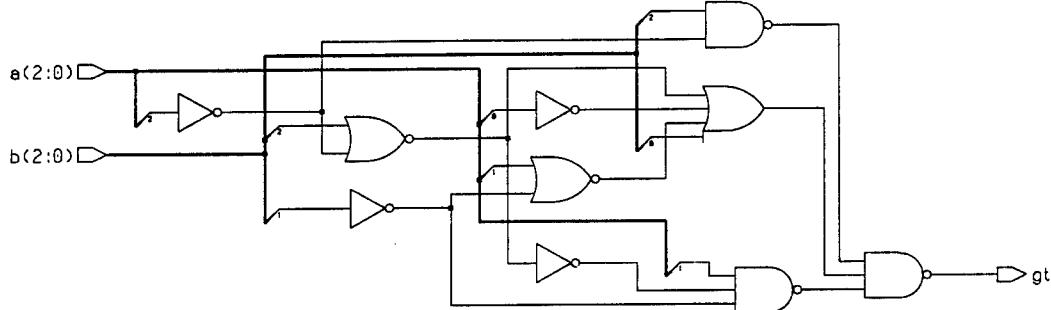


Figure A-10 Synthesized Structure1 for example GREATER3.

According to MIL-STD 883D, Structure1 contains 50 unique SSL gate level faults. Fault simulations using the behavioral test vectors from Figure A-9 resulted in complete gate level fault coverage.

Next, an alternate synthesis tool, Leonardo, was used to map the VHDL behavioral description to a Xilinx FPGA architecture. Fault simulations using the behavioral test vectors achieved a SSL gate level fault coverage of $48/48 = 100\%$.

Lastly, the signed comparison was implemented using *arithmetic operations*. Evaluation of the *greater than* function can also be performed by subtracting the two *operands* and examining the sign of the result. For example GREATER3, the most significant bit of the operation $B - A$ forms the output GT . The resulting synthesized and optimized circuit for Structure3 is shown in Figure A-11.

Comparison of the test vectors from Figure A-9 with the truth table for a full-subtractor from Table 5-15 indicates that the behavioral test vectors will provide complete func-

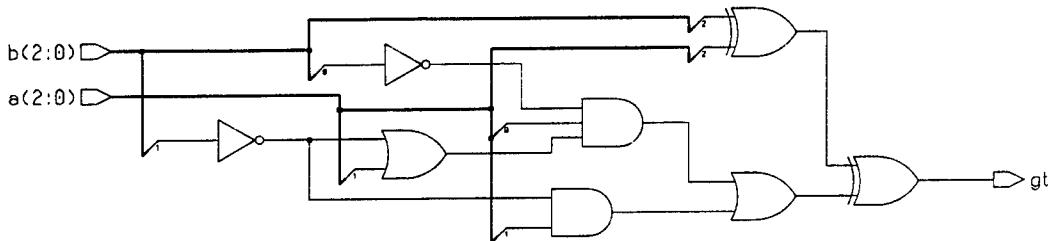


Figure A-11 Synthesized Structure3 for example GREATER3.

tional testing for the subtractor modules used to synthesize Structure3. Fault simulations confirm the complete gate level fault coverage of $41/41 = 100\%$.

A.4 Unsigned Threshold

Example GE23u is an unsigned threshold comparison using a *greater than or equal to (GE) operator*. As shown in Figure A-12, A is an integer with range 0 to 31, which will be synthesized to a 5-bit unsigned number.

```

entity GE23u is
    port(A: in INTEGER range 0 to 31;
         GE: out BIT);
end GE23u;
architecture BEHAVE of GE23u is
begin
    process(A)
        begin
            if A >= 23 then
                GE <= '1';
            else
                GE <= '0';
            end if;
        end process;
    end BEHAVE;
```

Figure A-12 Behavioral description for example GE23u.

The *GE* function places the threshold between 22 and 23. According to the behavioral fault model for *threshold operators*, developed in Chapter 4, the threshold lies seven spaces from the center of the range of values, implying step sizes of one, two, and four. Test vectors to the left of the threshold are $22 - 1 = 21$, $21 - 2 = 19$, and $19 - 4 = 15$. Test vectors to the right are $23 + 1 = 24$, $24 + 2 = 26$, and $26 + 4 = 30$. The behavioral test vectors are shown graphically in Figure A-13 and as a WAVES file in Figure A-14.

A >= 23	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
---------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Figure A-13 Behavioral test vectors for example GE23u.

```
% A GE : time;
01111 0 : 500 ns;
10001 0 : 500 ns;
10101 0 : 500 ns;
10110 0 : 500 ns;
10111 1 : 500 ns;
11000 1 : 500 ns;
11010 1 : 500 ns;
11110 1 : 500 ns;
```

Figure A-14 WAVES test vectors for example GE23u.

Example GE23u was synthesized to the gate level circuit shown in Figure A-15. Fault simulations using the behavioral test vectors from Figure A-14 resulted in a SSL gate level fault coverage of 24/24 = 100%.

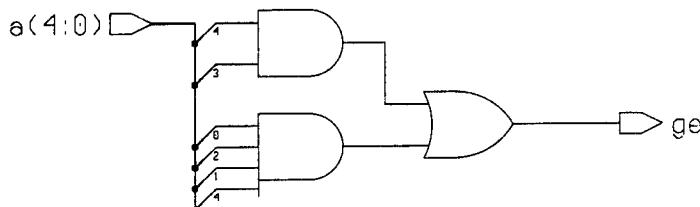


Figure A-15 Synthesized circuit for example GE23u.

A.5 Adder/Subtractor

Figure A-16 gives the VHDL behavioral description for a 4-bit adder/subtractor circuit, ADDSUB4. The two inputs (*A,B*) are combined to produce a 4-bit output (*D*). The operation to be performed is selected by the control signal (*OP*): *OP* = '0' selects addition, while *OP* = '1' selects subtraction.

```
if OP = '0' then
    D <= A + B;
else
    D <= A - B;
end if;
```

Figure A-16 Behavioral description for example ADDSUB4.

A.5.1 Faults on Arithmetic Operators

Assuming a simple *ripple carry* implementation, the behavioral test vectors for the 4-bit addition and 4-bit subtraction can be determined directly from the fault models. The resulting WAVES file is presented in Figure A-17.

```
% OP A   B   D : time;
% ADD
0 0000 0000 0000 : 500 ns;
0 0000 1111 1111 : 500 ns;
0 1111 0000 1111 : 500 ns;
0 1111 1111 1110 : 500 ns;
0 0001 1111 0000 : 500 ns;
0 1111 0001 0000 : 500 ns;
0 0101 0101 1010 : 500 ns;
0 1010 1010 0100 : 500 ns;
% SUB
1 0000 0000 0000 : 500 ns;
1 0000 1111 0001 : 500 ns;
1 1111 0000 1111 : 500 ns;
1 1111 1111 0000 : 500 ns;
1 0000 0001 1111 : 500 ns;
1 1110 1111 1111 : 500 ns;
1 0101 1010 1011 : 500 ns;
1 1010 0101 0101 : 500 ns;
```

Figure A-17 WAVES test vectors for example ADDSUB4.

A.5.2 Control Faults

The control fault model specifies that each clause of an *if-then-else* statement can be affected by two different types of faults, *Clause-CORRUPT (OR)* and *Clause-CORRUPT (AND)*. The behavioral test vectors for the *arithmetic operators* must be evaluated to ensure that adequate coverage is provided for these control faults.

For example, the first control fault, *THEN-CORRUPT (OR)*, results in the corrupted version of the *then* clause:

$$D \leq (A + B) \text{ OR } (A - B)$$

To test for this fault, the uncorrupted version of the clause ($A + B$) needs to be set to 0, while the corrupting clause ($A - B$) is set to 1. The Phase II test vector $A = 1111, B = 0001$ results in $A + B = 0000$, while $A - B = 1110$. Since no combination of A, B will produce

complementary values for D_0 , this test vector should provide sufficient coverage for this control fault. Other control faults and their covering test vectors are summarized in Table A-3. Note that no additional test vectors are required to provide coverage for all control faults.

Control Fault	OP	A	B	$A + B$	$A - B$
THEN-CORRUPT (OR)	0	1111	0001	0000	1110
THEN-CORRUPT (AND)	0	0000	1111	1111	0001
ELSE-CORRUPT (OR)	1	1111	1111	1110	0000
ELSE-CORRUPT (AND)	1	0000	0001	0001	1111

Table A-3 Coverage for control faults.

A.5.3 Evaluation of the Behavioral Test Vectors

Example ADDSUB4 was synthesized to the gate level circuit in Figure A-18. Fault simulation using the behavioral test vectors from Figure A-17 results in complete gate level fault coverage of $124/124 = 100\%$.

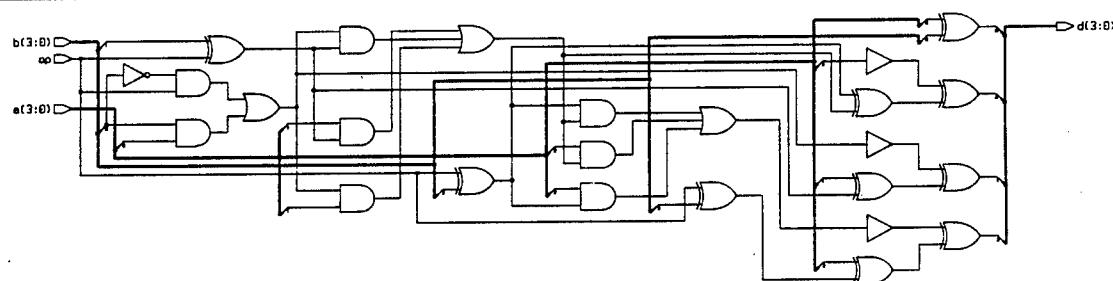


Figure A-18 Synthesized circuit for example ADDSUB4.

A.5.4 CLA Implementation

If the target technology includes *CLA* circuits, additional test vectors are required to cover the behavioral faults. The vectors for the addition operation are those derived in Chapter 5 and summarized in Table 5-10. Behavioral test vectors for the subtraction operation can be derived using 2's complement arithmetic.

The highest order carry in a 4-bit subtractor implemented with a *CLA* adder is C_2 . From Table 5-7, the terms for C_2 , including a *carry-in*, are $P_2P_1P_0C_{-1}$, $P_2P_1G_0$, P_2G_1 , and

G_2 . Behavioral test vectors for *missing carry* and *extra carry* faults can be derived for the subtraction operation by converting the corresponding addition to subtraction.

According to the new behavioral fault model, test vectors are necessary for *missing carry* faults for all but the lowest and highest order terms. *Missing carry* behavioral faults are presented in Table A-4. The left hand side of the table gives the test vector for addition, while the right hand side shows the conversion to subtraction.

Stage	Term	Addition				Subtraction		
		A	B	CIN	S	M	S	D
0	$P_2P_1G_0$	0001	1111	1	0001	0001	0000	0001
1	P_2G_1	0010	1111	1	0010	0010	0000	0010

Table A-4 Missing carry faults for subtraction.

Test vectors for *extra carry* faults are required for all but the two most significant stages. Behavioral test vectors for the *extra carry* faults are presented in Table A-5. The *CLA* test vectors for example ADDSUB4 are, therefore, presented in Figure A-19.

Stage	Term	Addition				Subtraction		
		A	B	CIN	S	M	S	D
-1	$P_2P_1P_0C_{-1}$	0000	1110	1	1111	0000	0001	1111
		0000	1101	1	1110	0000	0010	1110
		0000	1011	1	1100	0000	0100	1100
0	$P_2P_1G_0$	0001	1101	1	1111	0001	0010	1111
		0001	1011	1	1101	0001	0100	1101

Table A-5 Extra carry faults for subtraction.

A *CLA* implementation of example ADDSUB4 is presented in Figure A-20 [41]. Fault simulations were performed using the *ripple carry* test vectors from Figure A-17, plus the *CLA* test vectors from Figure A-19. The *ripple carry* vectors alone produced a *SSL* gate level fault coverage of $203/209 = 97.13\%$. The remaining faults were then detected by the *CLA* test vectors, resulting in complete gate level fault coverage.

```
% OP A B D : time;
% ADD CLA
0 0001 0011 0100 : 500 ns;
0 0001 0101 0110 : 500 ns;
0 0010 0110 1000 : 500 ns;
% SUB CLA
1 0001 0000 0001 : 500 ns;
1 0010 0000 0010 : 500 ns;
1 0000 0001 1111 : 500 ns;
1 0000 0010 1110 : 500 ns;
1 0000 0100 1100 : 500 ns;
1 0001 0010 1111 : 500 ns;
1 0001 0100 1101 : 500 ns;
```

Figure A-19 CLA test vectors for example ADDSUB4.

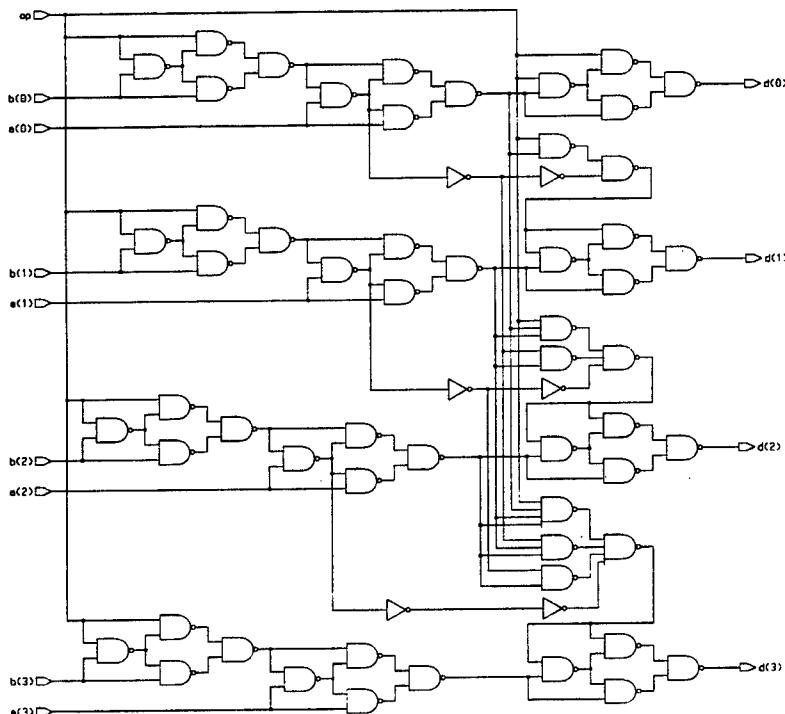


Figure A-20 CLA implementation of example ADDSUB4.

A.6 Arithmetic with Constants

The behavioral test vector patterns developed in Chapter 5 will now be applied to two larger examples to demonstrate the effect of synthesis optimizations on gate level fault coverage. First, example PLUS25 adds the *constant* 25 to an 8-bit number. Next, example MINUS25 combines the *subtraction operator* with the same *constant*. Alternately, this example can be viewed as *addition* using the *constant* -25.

A.6.1 Example PLUS25

Example PLUS25 implements the arithmetic operation $Z \leq Y + 25$. Signals Y and Z are declared with ranges from -128 to +127, hence they will be synthesized as 8-bit 2's complement numbers. The binary representation for +25 is 00011001, thus the Phase II test patterns are formed starting with the vector 11100110. The resulting behavioral test vectors are shown in Figure A-21.

```
% Y Z : time;
00000000 00011001 : 500 ns;
11111111 00011000 : 500 ns;
%
11100110 11111111 : 500 ns;
11001101 11100110 : 500 ns;
10011011 10110100 : 500 ns;
00110111 01010000 : 500 ns;
01101110 10000111 : 500 ns;
11011100 11110101 : 500 ns;
10111001 11010010 : 500 ns;
01110011 10001100 : 500 ns;
```

Figure A-21 Behavioral test vectors for example PLUS25.

Example PLUS25 was synthesized and optimized to the gate level Structure1 shown in Figure A-22. Note that the optimization process has altered the original *ripple carry* structure of the 8-bit adder.

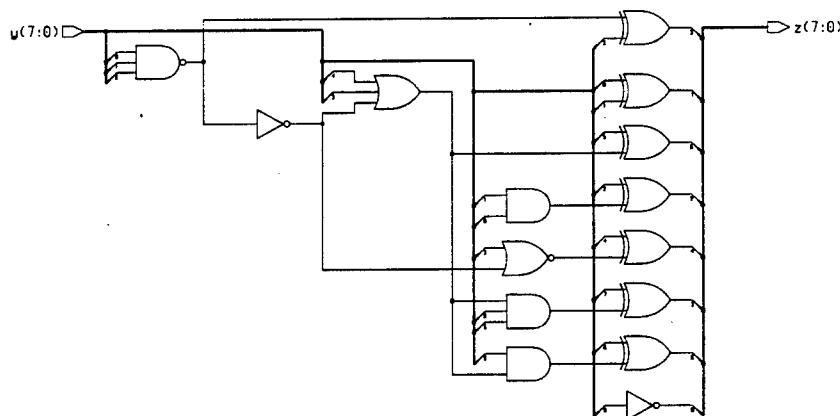


Figure A-22 Synthesized Structure1 for example PLUS25.

Fault simulations using the behavioral test vectors from Figure A-21 resulted in a *SSL* gate level fault coverage of $114/115 = 99.13\%$. The functional testing approach is still

able to achieve near complete fault coverage. An alternate synthesis tool and target architecture was next used to produce Structure2 for example PLUS25. Fault simulations using the behavioral test vectors from Figure A-21 resulted in a SSL gate level fault coverage of $190/190 = 100\%$.

A.6.2 Example MINUS25

Example MINUS25 performs the arithmetic operation $Z \leq Y - 25$. Again, the binary representation for +25 is *00011001*, resulting in the behavioral test vectors and associated functional tests shown in Table A-6.

Phase	Y	Z	Test #						
			ST ₇	ST ₆	ST ₅	ST ₄	ST ₃	ST ₂	ST ₁
I	00000000	11100111	0	0	0	0	0	0	0
	11111111	11100110	3	3	3	3	3	3	3
II	00011001	00000000	1	1	1	3	3	1	1
	00110010	00011001	1	1	2	2	1	1	2
	01100100	01001011	1	3	2	0	1	2	0
	11001000	10101111	3	2	0	0	2	0	0
	10010001	01111000	2	0	0	2	1	1	1
	00100011	00001010	1	1	2	0	1	1	3
	01000110	00101101	1	2	0	0	1	3	2
	10001100	01110011	2	0	0	1	3	2	0

Table A-6 Functional tests for example MINUS25.

Example MINUS25 was synthesized and optimized using the same process as Structure1 for example PLUS25. Fault simulations using the behavioral test vectors from Table A-6 again resulted in a SSL gate level fault coverage of $114/115 = 99.13\%$.

A.7 XOR4

Using the behavioral fault models developed in Chapter 6, test vectors can be developed for example XOR4 shown below.

$$Z \leq A \text{ XOR } B \text{ XOR } C \text{ XOR } D$$

First, the parse tree method from Section 6.1.1.2 will identify the test vectors necessary for exhaustive testing of each *XOR operator*. Next, the modified Bossen algorithm from Section 6.1.2.2 will produce an optimized set of test vectors for the given *expression*.

A.7.1 Parse Tree Test Vectors

A parse tree for example XOR4 is shown in Figure A-23. The nodes (1,2,3) are formed by the *XOR operators*, while the leaves of the tree are the signals *A*, *B*, *C*, and *D*.

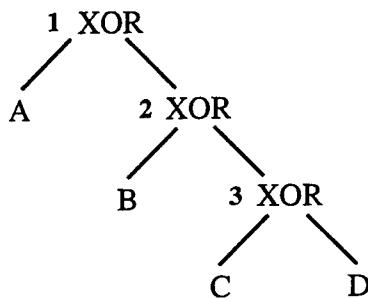


Figure A-23 Parse tree for example XOR4.

According to the generalized set of functional faults from Section 6.1.2.1, an exhaustive set of four tests is necessary for each *XOR operator*. Applying these tests to the parse tree from Figure A-23 produces the behavioral test vectors shown in Table A-7.

Node	Behavioral Test	Test Requirements	Test Vector (ABCD)
1	00	$A = 0, B \text{ XOR } C \text{ XOR } D = 0$	0000
	01	$A = 0, B \text{ XOR } C \text{ XOR } D = 1$	0100
	10	$A = 1, B \text{ XOR } C \text{ XOR } D = 0$	1000
	11	$A = 1, B \text{ XOR } C \text{ XOR } D = 1$	1100
2	00	$B = 0, C \text{ XOR } D = 0, A = 0$	0000
	01	$B = 0, C \text{ XOR } D = 1, A = 0$	0010
	10	$B = 1, C \text{ XOR } D = 0, A = 0$	0100
	11	$B = 1, C \text{ XOR } D = 1, A = 0$	0110

Table A-7 Behavioral test vectors for example XOR4.

Node	Behavioral Test	Test Requirements	Test Vector (ABCD)
3	00	C = 0, D = 0, B = 0, A = 0	0000
	01	C = 0, D = 1, B = 0, A = 0	0001
	10	C = 1, D = 0, B = 0, A = 0	0010
	11	C = 1, D = 1, B = 0, A = 0	0011

Table A-7 Behavioral test vectors for example XOR4.

Eliminating redundant test vectors produces the WAVES file shown in Figure A-24.

```
% ABCD Z : time ;
0000 0 : 500 ns;
0001 1 : 500 ns;
0010 1 : 500 ns;
0011 0 : 500 ns;
0100 1 : 500 ns;
0110 0 : 500 ns;
1000 1 : 500 ns;
1100 0 : 500 ns;
```

Figure A-24 WAVES test vectors for example XOR4.

A.7.2 Evaluation of Behavioral Test Vectors

Example XOR4 was synthesized into multiple gate level realizations to evaluate the behavioral test vectors. Varying structures, synthesis tools, and design libraries were employed to produce a broad range of implementations.

Structure1 of example XOR4 is shown in Figure A-25. Fault simulations using the behavioral test vectors from Figure A-24 result in a SSL fault coverage of 24/24 = 100%.

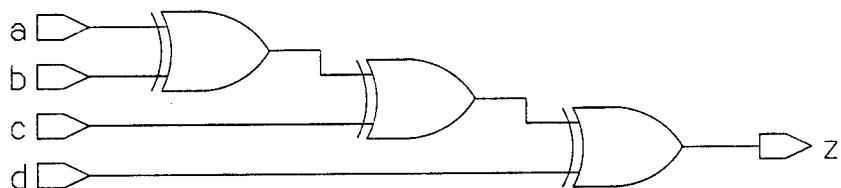


Figure A-25 Structure1 for example XOR4.

Structure 2 for example XOR4 is shown in Figure A-26. Fault simulations with the behavioral test vectors also produce a *SSL* gate level fault coverage of $24/24 = 100\%$.

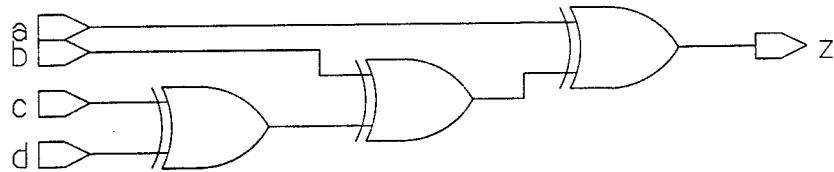


Figure A-26 Structure2 for example XOR4.

An alternate synthesis tool and target architecture were used to produce Structure3 for example XOR4, shown in Figure A-27. Fault simulations using the same behavioral test vectors from Figure A-24 result in a *SSL* gate level fault coverage of $54/54 = 100\%$.

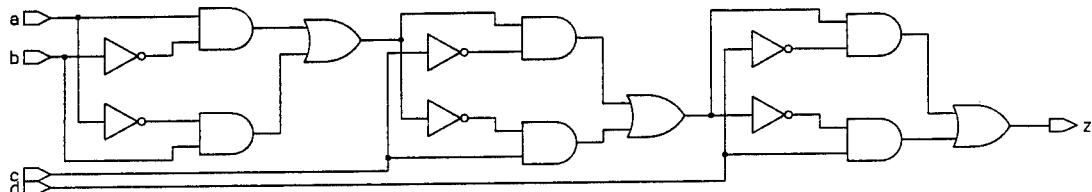


Figure A-27 Structure3 for example XOR4.

Lastly, Structure4 for example XOR4 is presented in Figure A-28. Again, fault simulations using the behavioral test vectors result in complete gate level fault coverage.

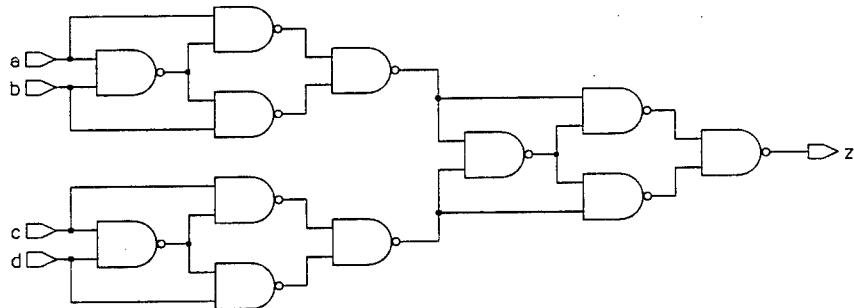


Figure A-28 Structure4 for example XOR4.

A.7.3 Optimized Test Vectors

Applying the modified Bossen algorithm to example XOR4 results in the labelling shown in Figure A-29 and Figure A-30. Note, the 4-bit example represents a special case where it is possible to find identical test sequences for structures Cascade1 and Cascade2.

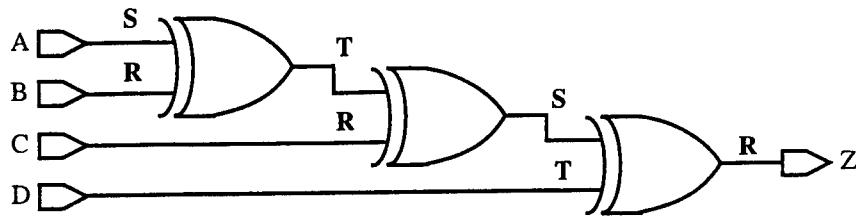


Figure A-29 Structure Cascade1 for example XOR4.

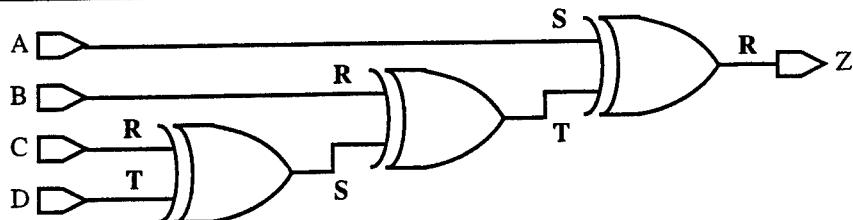


Figure A-30 Structure Cascade2 for example XOR4.

The resulting optimized test vectors are shown in Table A-8.

Signal	Label	Sequence	Test Vector (ABCD)	Z
A	S	0011	0000	0
B	R	0110	0111	1
C	R	0110	1110	1
D	T	0101	1001	0

Table A-8 Optimized test vectors for example XOR4.

A.7.4 Evaluation of Optimized Test Vectors

Fault simulations were conducted using the optimized behavioral test vectors from Table A-8 and Structures1-4 for example XOR4. For all four implementations of example XOR4, complete gate level fault coverage was achieved using either test vector set.

Appendix B

Fault Experiment Results

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
CASE1	Structure1	34	8	100%
	Structure2	34	8	100%
ARRAY4	Structure1	44	8	100%
	Structure2	44	8	100%
SHIFT4u	Structure1	90	8	100%
	Structure2	112	8	100%

Table B-1 Control fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
LESS2	Structure1	29	8	100%
	Structure2	30	8	100%
EQUAL3	Structure1	31	8	100%
	Structure2	31	8	100%
GREATER3	Structure1	50	18	100%
	Structure2	48	18	100%
	Structure3	41	18	100%
LE5	Structure1	18	6	100%
	Structure2	20	6	100%
GE23u	Structure1	24	8	100%
	Structure2	22	8	100%
LT12u	Structure	14	4	100%
GT3n	Structure	17	5	100%

Table B-2 Relational operator fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
COMPARE	Structure1	74	12	98.65%
			12	98.65%
	Structure2	72	12	100%
			12	100%
COMPARE3	Structure1	97	22	97.94%
			22	97.94%
	Structure2	92	22	100%
			22	100%
COMPARE4	Structure	150	12	100%
			12	100%
COMPARE34	Structure	178	22	100%
			22	100%

Table B-2 Relational operator fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
ADD4	Structure1	102	8	100%
	Structure2	130	8	97.69%
			11	100%
ADD4wc	Structure3	142	8	100%
	Structure1	138	8	100%
	Structure2	188	8	88.30%
			20	100%
	Structure3	194	8	100%

Table B-3 Arithmetic operator fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
ADD8	Structure1	234	8	100%
	Structure2	310	8	93.55%
			33	100%
			22	100%
SUB4	Structure1	112	8	100%
	Structure2	157	8	94.90%
			15	100%
ADDSUB4	Structure1	124	16	100%
	Structure2	209	26	100%
INC4	Structure1	50	6	100%
	Structure2	80	6	100%
INC8	Structure	114	10	100%
DEC4	Structure	51	6	100%
ADDINC4	Structure	108	14	100%
PLUS3	Structure	68	7	100%
MINUS5	Structure	85	8	100%
PLUS25	Structure1	115	10	99.13%
	Structure2	190	10	100%
MINUS25	Structure ..	115	10	99.13%

Table B-3 Arithmetic operator fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
SOP1	Structure1	18	5	100%
	Structure2	18	5	100%
SOP4	Structure1	72	5	100%
	Structure2	72	5	100%

Table B-4 Other operator fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
POS1	Structure1	18	5	100%
	Structure2	18	5	100%
GT	Structure1	30	8	100%
	Structure2	35	8	100%
	Structure3	29	8	100%
XOR4	Structure1	24	8	100%
			4	100%
	Structure2	24	8	100%
			4	100%
	Structure3	54	8	100%
			4	100%
	Structure4	44	8	100%
			4	100%
XOR5	Structure1	30	7	100%
	Structure2	30	7	100%
	Structure3	58	7	100%
	Structure4	58	7	100%
ABS4	Structure1	44	5	100%
	Structure2	44	5	100%
ABS8	Structure1	132	9	100%
	Structure2	126	9	100%
NEG4	Structure1	49	5	100%
	Structure2	50	5	100%
NEG8	Structure1	114	9	100%
	Structure2	204	9	99.02%

Table B-4 Other operator fault experiments.

Example	Implementation	SSL Faults	Behavioral Test Vectors	Fault Coverage
ALU4wc	Structure1	284	31	100%
	Structure2	312	31	100%
	Structure3	398	31	100%
ALU8wc	Structure1	546	31	100%
	Structure2	758	31	100%
HAMMING4	Structure1	114	17	100%
	Structure2	116	17	100%
	Structure3	226	17	100%
HAMMING8	Structure1	226	33	100%
	Structure2	446	33	100%

Table B-5 Comprehensive fault experiments.

Appendix C

VHDL Behavioral Descriptions

VHDL can be used to model the function performed by a module at a level of abstraction above the gate level. Such a description is called a functional or behavioral description. *The VHDL Cookbook* [11] and the *IEEE Standard VHDL Language Reference Manual* [38] serve as two key resources for designing hardware using behavioral descriptions. VHDL allows description of behavior in the form of a sequence of familiar programming language constructs. Behavioral descriptions use *variables* and *signals* along with their corresponding *assignment statements* to model the desired functionality of the ultimate hardware. *Expressions* perform arithmetic or logical computations by applying an *operator* to one or more *operands*. Constructs used to control the selection and sequencing of instructions include *if*, *case*, and *loop* statements. Concurrency of execution in hardware is modeled using a *process* statement. Like other programming languages, VHDL provides subprogram facilities in the form of *procedures* and *functions*.

C.1 Variables and Signals

An object is a named item in a VHDL description which has a value of a specified type. A *variable* is an object whose value may be changed at any time during the simulation of the circuit. It is local to a process or subprogram and has a single current value. A *signal* is an object with a value that is changed only at scheduled times. Signals represent electrical quantities that can be used to transmit information and are normally used to connect submodules in a design. Each object has a corresponding *assignment statement*.

As in other programming languages, a *variable* is given a new value using an assignment statement. In the simplest case, the target of the assignment is an object name and the value of the expression is given to the named object. *Variable* assignments occur immediately when the assignment statement is executed and are local to a *process* or subprogram. A *signal* assignment schedules a transaction to a signal. The target must represent a *signal* or be an aggregate of *signals*. Scheduled transactions are executed as simulation time progresses. *Signals* are global in a *process* or subprogram and are the only means of communication between *processes*.

C.2 Expressions

An *expression* is a formula that defines the computation of a value. The type of an *expression* depends only upon the types of its *operands* and on the *operators* applied. The predefined VHDL *operators* are listed in Table C-1.

Type	Operators					
Logical	<i>AND</i>	<i>OR</i>	<i>NAND</i>	<i>NOR</i>	<i>XOR</i>	
Relational	=	/=	<	<=	>	>=
Adding	+	-	&			
Unary (sign)	+	-				
Multiplying	*	/	<i>MOD</i>	<i>REM</i>		
Miscellaneous	**	<i>ABS</i>	<i>NOT</i>			

Table C-1 Predefined VHDL operators.

Relational *operators* must have both *operands* of the same type and yield Boolean results. The equality *operators* (= and /=) can have *operands* of any type. The remaining *operators* must have *operands* which are scalar types or one-dimensional arrays of discrete types.

The sign *operators* (+ and -) and the addition (+) and subtraction (-) *operators* have their usual meaning on numeric *operands*. The concatenation *operator* (&) operates on one-dimensional arrays to form a new array with the contents of the right *operand* following the contents of the left *operand*. It can also concatenate a single new element to an array or two individual elements to form an array.

The multiplication (*) and division (/) *operators* work on integer and floating point types. The modulus (*MOD*) and remainder (*REM*) *operators* only work on integer types. The absolute value (*ABS*) *operator* works on any numeric type. Finally, the exponentiation (**)*operator* can have an integer or floating point left *operand*, but must have an integer right *operand*.

C.3 If Statement

The *if* statement allows selection of statements to execute depending on one or more conditions. The syntax is:

```

if_statement ::=

  if condition then
    sequence_of_statements
  { elseif condition then
    sequence_of_statements }
  [ else
    sequence_of_statements ]
end if;

```

The conditions are *expressions* resulting in Boolean values. The conditions are evaluated successively until one is found that yields the value *true*. In that case the corresponding statement list is executed. Otherwise, if the *else* clause is present, its statement list is executed.

C.4 Case Statement

The *case* statement allows selection of statements to execute depending on the value of a selection *expression*. The syntax is:

```

case_statement ::=

  case expression is
    case_statement_alternative
    { case_statement_alternative }
  end case;

case_statement_alternative ::=

  when choices =>
    sequence_of_statements

choices ::= choice { | choice }

choice ::=

  simple_expression
  | discrete_range
  | element_simple_name
  | others

```

The selection *expression* must result in either a discrete type, or a one-dimensional array of characters. The alternative whose *choice* list includes the value of the *expression* is selected and the statement list executed. Note that all the *choices* must be distinct, that is, no value may be duplicated. Furthermore, all values must be represented in the *choice*

lists, or the special choice *others* must be included as the last alternative. If no *choice* list includes the value of the *expression*, the *others* alternative is selected.

Some examples of *case* statements [11]:

```

case opcode is
    when X"00" => perform_add;
    when X"01" => perform_subtract;
    when others => signal_illegal_opcode;
end case;

case element_color is
    when red =>
        statements for red;
    when green | blue =>
        statements for green or blue
    when orange to turquoise =>
        statements for these colors;
end case;

```

C.5 Loop Statements

VHDL has a basic *loop* statement, which can be augmented to form the usual *while* and *for* loops seen in other programming languages. The *while* iteration scheme allows a test condition to be evaluated before each iteration. The iteration only proceeds if the test evaluates to *true*. If the test is *false*, the *loop* statement terminates. An example [11]:

```

while index < length loop
    index := index + 1;
end loop;

```

The *for* iteration scheme allows a specified number of iterations. The *loop* parameter specification declares an object which takes on successive values from the given range for each iteration of the *loop*. Within the statements enclosed in the *loop*, the object is treated as a constant, and so may not be assigned to. An example [11]:

```

for item in 1 to last_item loop
    table(item) := 0;
end loop;

```

There are two additional statements which can be used inside a *loop* to modify the basic pattern of iteration. The *next* statement terminates execution of the current iteration

and starts the subsequent iteration. The *exit* statement terminates execution of the current iteration and terminates the *loop*. The syntax of these statements is:

```
next_statement ::= next [ loop_label ] [ when condition ];
exit_statement ::= exit [ loop_label ] [ when condition ];
```

If the *loop* label is omitted, the statement applies to the inner-most enclosing *loop*, otherwise it applies to the named *loop*. If the *when* clause is present but the condition is *false*, the iteration continues normally.

C.6 Process Statement

The primary unit of behavioral description in VHDL is the *process*. A *process* is a sequential body of code which can be activated in response to changes in state specified by a sensitivity list or a *wait* statement. When more than one *process* is activated at the same time, they execute concurrently.

An example of a *process* statement with a sensitivity list [11]:

```
process (reset, clock)
  variable state : bit := false;
begin
  if reset then
    state := false;
  elsif clock = true then
    state := not state;
  end if;
  q <= state after prop_delay;
  -- implicit wait on reset, clock
end process;
```

During the initialization phase of simulation, the *process* is activated and assigns the initial value of state to the signal *q*. It then suspends at the implicit *wait* statement indicated in the comment. When either reset or clock change value, the *process* is resumed and execution repeats from the beginning.

Processes, like all other concurrent statements, read and write *signals* and interface port values to communicate with the rest of the architecture. They are unique in that they behave like concurrent statements to the rest of the design, but they are internally sequential. In addition, only *processes* define *variables* to hold intermediate values in a sequence of computations.

C.7 Procedures and Functions

Subprograms, like *processes*, use sequential statements to define algorithms that compute values. Unlike *processes*, however, they cannot directly read or write *signals* from the rest of the architecture. All communication is performed through the subprogram's interface; each subprogram call has its own set of interface *signals*.

The two types of subprograms are *functions* and *procedures*. A *function* returns a single value directly. A *procedure* returns zero or more values through its interface. Subprograms may perform repeated calculations, often in different parts of an architecture. The syntax is:

```
subprogram_specification ::=  
    procedure designator [(parameter_list)] |  
    function designator [(parameter_list)] return type
```

By using these programming language constructs, designers can use VHDL to develop behavioral level models that can be simulated to verify their correct functioning, prior to generating hardware.

Appendix D

VHDL Synthesis

Automated design tools are available which allow the designer to synthesize and optimize circuit descriptions expressed through hardware description languages such as VHDL. In order to ensure proper and consistent synthesis of VHDL language constructs to hardware, such tools include guidelines which describe both the syntax and semantics of the subset of the entire VHDL language which they support. The VHDL Synthesis Interoperability Working Group (SIWG) was established to develop a public domain VHDL Synthesis Interoperability Standard which, if used by designers, will allow VHDL synthesis models to be portable across synthesis vendors that support this standard. Their synthesis domain is register transfer level (RTL) logic synthesis which corresponds to the functional/behavioral descriptions modeled in this and previous research.

A draft IEEE standard [36] has been developed which encompasses recommendations drawn from several key sources, including Cadence [43], Mentor Graphics [68], and Synopsys [64]. The foundation for the synthesis standard was laid by the European VHDL Synthesis Working Group's *Level-0 VHDL Synthesis Syntax and Semantics* [25]. Beyond the baseline Level-0, the draft proposes standardization Level 1 containing constructs that are currently supported by many synthesis tools. The standard also attempts to describe VHDL constructs as:

- 1) Supported: RTL synthesis will map the construct to hardware.
- 2) Ignored: RTL synthesis will ignore the construct. Encountering the construct will not cause synthesis to fail, but synthesis results may not match simulation results.
- 3) Not supported: RTL synthesis does not support the construct.

A synthesis tool is defined as any system, process, or tool that interprets register transfer level VHDL source code as a description of an electronic circuit in accordance with the terms of this standard and derives a gate level netlist description of that circuit.

D.1 Level-0

The *Level-0 VHDL Synthesis Syntax and Semantics* [25] represents a first step towards a standard for VHDL allowing its use in other hardware related tasks like formal verifica-

tion, fault simulation, test generation, etc. which currently can not be performed in a standard way. The intention behind the definition of Level-0 was to overcome numerous problems currently associated with high level synthesis. The Level-0 syntax and semantics constitute a standard subset of VHDL for synthesis applications which will allow description portability between tools as well as design reusability.

Level-0 represents the minimum syntactical and semantical requirements to any synthesis tool in the market. Of the 217 syntax elements described in Appendix A of the *VHDL Language Reference Manual* [38]:

- 106 syntax elements are fully supported.
- 53 syntax elements are supported with restrictions.
- 56 syntax elements are not allowed.
- 2 syntax elements are ignored.

Despite the current limitations imposed by the state of the art in commercial synthesis technology, the Level-0 synthesis syntax and semantics allows the description of digital systems as an interconnection of combinational and sequential blocks and, therefore, allows the description of any digital system at the RT level (i.e. algorithmic finite state machines). As a consequence, any synchronous VHDL description in any proprietary VHDL subset can be translated to the Level-0 synthesis syntax and semantics maintaining all the relevant information about the functionality of the design. This portable description will be accepted by any other synthesis tool giving functional equivalent results.

In addition to the support of specific language constructs, Level-0 also defines certain synthesis semantics and usage guidelines. Probably the most important of these design guidelines are those for processes, which serve as the basis for behavioral descriptions. Processes must have a set of special characteristics, which can be summarized in four different kinds of processes shown in Figure D-1.

Other restrictions imposed by Level-0 include the limited set of operators allowed. The operators: *abs*, “**”, “/”, *mod*, and *rem* are not supported. The multiplying operator “*” is supported, only if both operands are constants or the second operand is a power of two. Additionally, floating point operands are not allowed and the only type of loops allowed are *for* loops.

-
- 1) Processes which contain a sensitivity list including all the signals which are read into the process and in which all signals and variables are assigned in all the conditional branches. This kind of process models pure combinational logic.
 - 2) Processes which contain a sensitivity list including all the signals that are read into the process and whose variables are assigned in all the conditional branches of the process. This kind of process can model a mixture of pure combinational logic and asynchronous latches. Latches are inferred when signals are not assigned in a conditional branch.
 - 3) Processes which have, as their first statement, a wait statement in the form:

wait until clock = value and clock'event;

This kind of process models a Moore synchronous sequential machine.

- 4) Processes which have a sensitivity list including the *clock* signal and optionally an asynchronous *reset* signal and an *if* statement controlled by the event and edge of the clock signal. Thus, this kind of process has the following syntax:

```
process (clk_name, reset_name)
  process_declarative_part
begin
  if (reset_name = value) then
    { signal_assignment_statement }
  elsif (clk_name = value and clk_name'event) then
    { signal_assignment_statement }
  end if;
end process;
```

Figure D-1 Process types for Level-0.

D.2 Mentor Graphics

The *VHDL Style Guide for AutoLogic II* [68] describes how to write synthesizable VHDL for the Mentor Graphics AutoLogic II synthesis environment. The guidelines encompass language restrictions, style issues, modeling methods, and design methods. Section 1 of the manual summarizes the subset of VHDL language supported for synthesis. All of the language elements previously discussed in Appendix C of this dissertation are supported by AutoLogic II. In addition to the predefined VHDL operators listed in

Table C-1, shift operators from VHDL 1993 are also supported. These include: shift left/right arithmetic (sla/sra), shift left/right logical (sll/srl), and rotate left/right (rol/ror).

AutoLogic II also supports the Mentor Graphics' std_logic_arith package, defined in the Mentor Graphics' ARITHMETIC library. This package includes functions that allow signed and unsigned math to be performed using the STD_LOGIC and STD_LOGIC_VECTOR types. The functions provide for binary and unary arithmetic operations beyond the numeric operands predefined by VHDL. The package also defines the SIGNED and UNSIGNED types that are used directly with arithmetic and relational shifts, comparison, logical, and type conversion functions. Relational operators are supplemented by the comparison functions: equal (*eq*), not equal (*ne*), less than (*lt*), greater than (*gt*), greater than or equal to (*ge*), and less than or equal to (*le*).

The remainder of the manual presents the VHDL constructs and modeling styles necessary to synthesize varying types of hardware. A generous set of examples provides the designer with a virtual cookbook on how to design specific circuits. Section 2 presents constructs and style issues describing hardware functionality implemented with a combinatorial architecture. Topics include signal assignments and sequential statements with special emphasis given to the control constructs *if* and *case*. Section 3 discusses how AutoLogic II recognizes clocked sections of descriptions and defines edge sensitive and level sensitive latch inferencing. The guidelines for combinational and sequential processes agree with those discussed in Figure D-1 as part of the Level-0 synthesis semantics.

Section 4 of the manual introduces the use of predefined operators and functions, data types, and procedures used in describing hardware with VHDL. Descriptions and examples give details for use of the predefined modeling environment, as well as specifics on the types of hardware to be synthesized. Finally, Section 5 presents the guidelines for developing and synthesizing synchronous state machine descriptions. Numerous examples describe multiple modeling styles for both Mealy and Moore class state machines.

D.3 Synopsys

Input to the VHDL Synthesis Interoperability Working Group was provided by Synopsys in the form of a *Standard for Synthesizing from VHDL Language at the Register Transfer Level* [64], which now takes the form of an unapproved IEEE Standards Draft.

This standard describes the use of a synthesis tool to translate high-level VHDL descriptions to gate-level netlists. The purpose of the standard is to define how a synthesis tool shall behave when synthesizing from VHDL at the register transfer level. Clauses 3 through 7 describe the language constructs, data types, expressions, sequential statements, and concurrent statements used for writing VHDL design descriptions. The standard is really just a language reference manual, with examples, for the subset of VHDL that is applicable to synthesis.

Many VHDL constructs, although useful for simulation and other stages in the design process, are not relevant to synthesis. Because these constructs cannot be synthesized, they are not supported by the synthesis tool. Clause 8 provides a list of all VHDL constructs with the level of support for each. A construct may be fully supported, ignored, or unsupported. Ignored means that the construct is allowed in the VHDL source, but is ignored by the synthesis tool. Unsupported means that the construct is not allowed in the VHDL source and that the synthesis tool flags it as an error.

Some of the key design restrictions imposed by the synthesis tool are summarized in Figure D-2. Though these restrictions give very specific guidance to the designer regard-

-
- 1) Integer types are automatically converted to bit vectors whose width is as small as possible to accommodate all possible values of the type's range, either in unsigned binary for nonnegative ranges or in 2's complement form for ranges that include negative numbers.
 - 2) Floating-point types, such as *REAL*, are unsupported.
 - 3) The arithmetic operators "+" and "-" are predefined for all integer operands. For adders more than four bits wide, a synthetic library component is used.
 - 4) Multiplying operators ("*", "/", mod, and rem) are predefined for all integer types with the following restrictions:
 - a) "*" (Integer multiplication) - No restrictions. A multiplication operator is implemented as a synthetic library cell.
 - b) "/" (Integer division) - The right operand shall be a power of 2. Neither operand shall be negative. Implemented as a bit shift.
-

Figure D-2 Design restrictions for Synopsys

-
- 5) Some forms of the *if* statement may be used to test for signal edges and, therefore, imply synchronous logic. This usage causes the synthesis tool to infer registers or latches.
 - 6) Synthesized hardware is sensitive to all signals read by a process. To guarantee that a VHDL simulator distinguishes the same results as the synthesized hardware, a process sensitivity list shall contain all signals whose changes require resimulation of that process.
-

Figure D-2 Design restrictions for Synopsys

ing use of the language, the details of the implementation by the synthesis tool are hidden. This lack of synthesis information makes the Synopsys tool less useful than Mentor Graphics' tool for exploring the relationship of language constructs with resulting hardware for the development of higher level fault models.

D.4 IEEE Draft Standard

The resulting *IEEE Draft Standard for VHDL Register Transfer Level Synthesis* [36] builds on Level-0 and incorporates constructs common to synthesis tools by Mentor Graphics and Synopsys. The draft standard represents a subset of VHDL with corresponding design guidelines meant to ensure consistent synthesis of gate level netlists from behavioral descriptions.

As with the other synthesis environments discussed in this dissertation, a syntax subset of VHDL is defined. The key language constructs supported for behavioral modeling are listed below:

- 1) *if* statement, *case* statement, *loop* statement.
- 2) *procedure*, *function*.
- 3) *constant*, *variable*, *signal*.
- 4) all predefined VHDL operators shown in Table C-1.

Design restrictions are consistent with those discussed for Synopsys in Figure D-2. The only iteration scheme supported for the *loop* statement is *for*.

The draft standard also contains a set of representative design examples whose intent is to specify certain prevalent modeling styles resulting in basic hardware elements like flip-flops, latches, etc. The specification of processes and resulting inferred logic are

consistent with the description of combinational and sequential logic for Level-0 shown in Figure D-1.

Use of this subset is meant enhance the portability of VHDL designs across synthesis tools conforming to the standard. It should also minimize the potential of functional simulation mismatches between models before they are synthesized and models after they are synthesized. It, therefore, should also serve as the basis for defining higher level fault models which have a closer relationship to resulting synthesized hardware.

Appendix E

Hardware Implementation of VHDL Constructs

Several VHDL language constructs lend themselves directly to hardware implementation with common functional modules such as multiplexers. By examining these language to hardware relationships, this thesis intends to build the foundation on which higher level fault models can be defined, that are more closely related to their underlying gate level counterparts. The necessary insights will be drawn from two resources which directly discuss the relationship between certain VHDL constructs and the ultimate hardware.

One discussion of hardware implementation of VHDL constructs comes from *Structured Logic Design with VHDL* by Armstrong and Gray [10]. In a section titled “Automated Synthesis of VHDL Constructs,” they demonstrate the relationship between multiplexers and language constructs that involve selection, like *if* and *case*. Another insight into the relationship between VHDL language constructs and hardware comes from the *VHDL Style Guide for AutoLogic II* by Mentor Graphics [68]. Again, the link is established between the control constructs *if* and *case* and the multiplexer functional building block.

E.1 Structured Logic Design

As part of their discussion of algorithmic synthesis, Armstrong and Gray present the concept of automatic translation of a representative sample of VHDL constructs into hardware. They concentrate on translations that are application independent, rather than ones from specified programming styles into restricted sets of hardware.

The first discussion involves constructs that involve selection of a specific element from a specified set. The *case* statement implies selection of one case from a specified set of cases. The *if...then...else* statement implies selection of the highest priority condition that is *true* from a prioritized list of conditions. Also, one element of a vector may be selected by specifying an index value. All of these statements involve selection and, therefore, exhibit the functionality of a multiplexer. Figure E-1 shows several examples of VHDL constructs that can be mapped to multiplexer elements [10].

```

package TYPES is
    attribute ENCODING: STRING;
    type ENUM is (A, B, C, D);
        attribute ENCODING of ENUM: type is "00 01 10 11"
end types;

use work.TYPES.all;
entity MUX is
    port (X, Y: in BIT;
          VECT: in BIT_VECTOR(3 downto 0);
          CHOICE: in ENUM;
          INDEX: in INGETER range 3 downto 0;
          Z1, Z2, Z3: out BIT);
end MUX;

architecture MUX_CONSTRUCTS of MUX is
begin
    MUX1: process (CHOICE, X, Y)
    begin
        case CHOICE is
            when A => Z1 <= X;
            when B => Z1 <= Y;
            when C => Z1 <= not X;
            when D => Z1 <= not Y;
        end case;
    end process MUX1;

    MUX2: process (X, Y, VECT)
    begin
        if X = '1' then
            Z2 <= VECT(3);
        elsif Y = '1' then
            Z2 <= VECT(2);
        else
            Z2 <= VECT(1) and VECT(0);
        end if;
    end process MUX2;

    MUX3: process (VECT, INDEX)
    begin
        Z3 <= VECT(INDEX);
    end process MUX3;
end MUX_CONSTRUCTS;

```

Figure E-1 VHDL constructs that map to multiplexer elements.

Process MUX1 in Figure E-1 is an example of a *case* statement that can be implemented with a multiplexer element. The *case* statement selects a function of inputs *X* and *Y* to assign to output *Z1* based on the value of enumerative data type *CHOICE*. In package TYPES an attribute ENCODING of type STRING is declared which directs the automated design tool to assign binary codes to elements of the type ENUM. With this information, the tool translates the process MUX1 into the hardware circuit shown in Figure E-2. For example, when *CHOICE*=(00), *Z1*=*X* as implied by the VHDL source code.

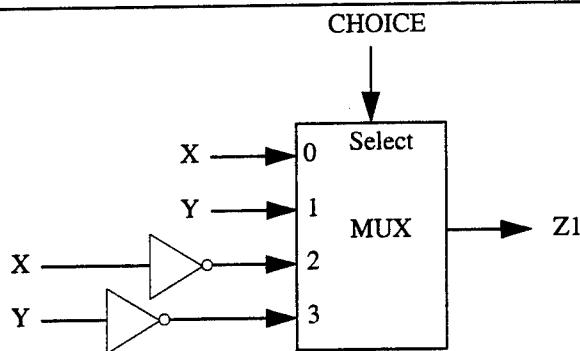


Figure E-2 Hardware implementation for *case* statement.

The VHDL construct *if...then...else* also involves selection among several alternative actions. Therefore, multiplexer elements can be used to implement this construct. Process MUX2 in Figure E-1 shows an example that involves inputs *X*, *Y*, and *VECT*. By scanning the *if...then...else* clause, the automatic design tool can produce the truth table shown in Table E-1. Notice that the first *if* clause that is *true* selects the action to be performed. It is possible that more than one *if* clause is *true*. For example, if *X=Y=1*, then two of the *if* clauses are *true*. However, in this case, *Z2* is assigned the value *VECT(3)* because the clause (*if X='1'*) takes precedence over the clause (*if Y='1'*).

X	Y	Z2
0	0	VECT(1) and VECT(0)
0	1	VECT(2)
1	0	VECT(3)
1	1	VECT(3)

Table E-1 Truth table for process MUX2.

Table E-1 directly implies the multiplexer implementation in Figure E-3, where signals X and Y are connected to the address inputs of the multiplexer and the data inputs for each XY combination are specified by the table entries.

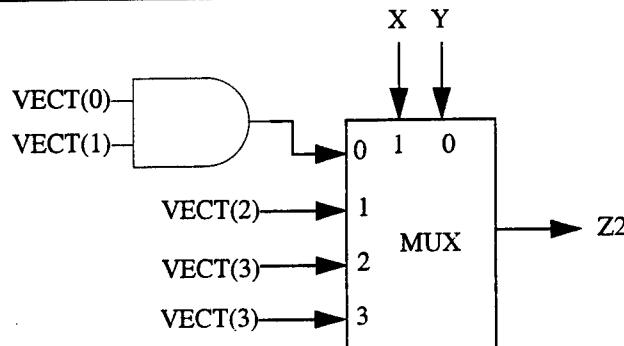


Figure E-3 Hardware implementation for *if* statement.

Finally, if $VECT$ is a vector and $INDEX$ is an integer, then an assignment of the form

$Z3 \leq VECT(INDEX)$

is also a selection activity. In this case, one of the elements of $VECT$, as specified by $INDEX$, is being assigned to $Z3$. This type of statement also maps directly to a multiplexer device. Process MUX3 in Figure E-1 shows an example of this type of selection activity. In the absence of an attribute specifying a coding other than binary for $INDEX$, the example leads directly to the circuit in Figure E-4.

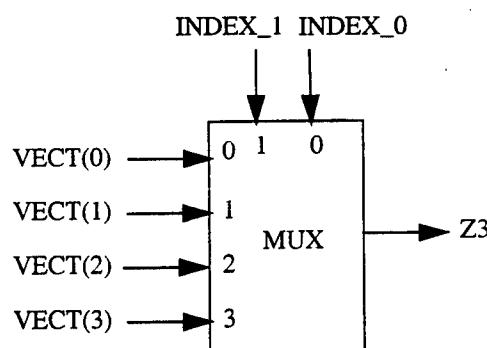


Figure E-4 Hardware implementation for vector indexing.

Next, program *loops* are discussed and illustrated through a classic ripple carry adder circuit. Figure E-5 shows the VHDL code for a 4-bit adder circuit implemented as a connection of full adders (*FA*). It is assumed that the *process* in architecture LOOP_ADDER

is embedded in a larger system that is not shown. The *loop* architecture maps directly to the iterative combinational logic network shown in Figure E-6.

```

entity ADD4 is
    port (A,B: in BIT_VECTOR(3 downto 0); CIN: in BIT;
          S: out BIT_VECTOR(3 downto 0); COUT: out BIT);
end ADD4;

architecture LOOP_ADDER of ADD4 is
begin
    process (A, B, CIN)
        variable CARRY: BIT_VECTOR(4 downto 0) := "00000";
        variable SUM: BIT_VECTOR(3 downto 0);
    begin
        CARRY(0) := CIN;
        for I in 0 to 3 loop
            SUM(I) := A(I) xor B(I) xor CARRY(I);
            CARRY(I+1) := (A(I) and B(I)) or (A(I) and
                CARRY(I)) or (B(I) and CARRY(I));
        end loop;
        S <= SUM;
        COUT <= CARRY(4);
    end process;
end LOOP_ADDER;

```

Figure E-5 VHDL description for a ripple carry adder.

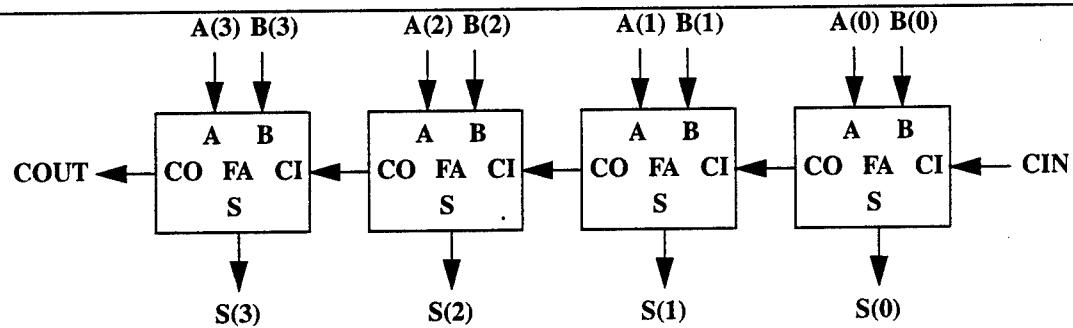


Figure E-6 Hardware implementation of ripple carry adder.

Lastly, Armstrong and Gray also illustrate possible mappings of *functions* and *procedures* to hardware. Figure E-7 shows VHDL code for architecture FUNCTION_ADDER of entity ADD4. In the example, the logic equations for a full adder (*FA*) are implemented by *function* declarations. There are separate declarations for each output of the *FA*: sum (*FA_S*) and carry (*FA_C*).

```

architecture FUNCTION_ADDER of ADD4 is
    function FA_S (AIN, BIN, CIN: BIT) return BIT is
    begin
        return AIN xor BIN xor CIN;
    end FA_S;
    function FA_C (AIN, BIN, CIN: BIT) return BIT is
    begin
        return (AIN and BIN) or (AIN and CIN) or
            (BIN and CIN);
    end FA_C;

begin
    process (A, B, CIN)
        variable CARRY: BIT_VECTOR(4 downto 0) := "00000";
        variable SUM: BIT_VECTOR(3 downto 0) := "0000";
    begin
        CARRY(0) := CIN;
        for I in 0 to 3 loop
            SUM(I) := FA_S(A(I), B(I), CARRY(I));
            CARRY(I+1) := FA_C(A(I), B(I), CARRY(I));
        end loop;
        S <= SUM;
        COUT <= CARRY(4);
    end process;
end FUNCTION_ADDER;

```

Figure E-7 Using functions to represent combinational logic.

Inside the program loop, the assignments to $SUM(I)$ and to $CARRY(I+1)$ are replaced by function calls. Since only the notation has changed, not the basic operation of the algorithm, it is clear that the architecture FUNCTION_ADDER can be mapped to the same hardware as architecture LOOP_ADDER. The general conclusion is that *functions* should be mapped to combinational logic circuits.

Similarly, *procedures* are used mainly as a convenience for ease of programming and could be used to replace the full adder in the previous example. In general, any VHDL code that uses procedures can be mapped to the same hardware as equivalent code without procedures. The main difference between procedures and functions is that functions always map to combinational logic, whereas procedures may map to sequential logic.

E.2 Mentor Graphics

The *VHDL Style Guide for AutoLogic II* [68] not only describes how to write synthesizable VHDL, but also gives details on how specific language constructs are implemented in the ultimate hardware. Examples are provided for concurrent *signal* assignment statements, *if* statements, *case* statements, and *variable* index assignments.

A concurrent *signal* assignment always results in combinational logic. This *signal* assignment specifies that a target *signal* is to receive some waveform. The waveform can either be a static value or some defined behavior. An example of a static assignment is:

```
D <= A;
```

This statement implies that *signal D* gets *signal A* and is therefore hard-wired to *A*. The waveform assigned to the target *signal* can also define some behavior. This behavior may be a simple logical *expression* or a complex *expression* that includes a *function* or *procedure* call. An example of a simple logical *expression* is:

```
D <= A AND B;
```

Assuming *D*, *A*, and *B* are all bits and ports, this description synthesizes into an *AND* gate.

The simple *if* statement does not contain enough information to synthesize a combinatorial network. For example, what happens if the condition evaluates to a boolean *false* value as described in Figure E-8. The output *signal D* receives input *signal B* when *A* is a '1'. This implies that if *signal A* is not a '1' then *signal D* must retain its old value. The description is then synthesized to a level-sensitive or transparent latch enabled by *signal A* whose output is *signal D*.

```
architecture RTL of IF_TEST is
begin
    process (A, B)
    begin
        if (A = '1') then
            D <= B;
        end if;
    end process;
end RTL;
```

Figure E-8 Code example for simple *if* statement.

In order to synthesize a combinatorial network using an *if* statement, the *if* statement must explicitly define the behavior for all possible evaluations of the condition. In Figure

E-8 this means that the *if* statement must define what signal *D* is to receive when signal *A* is not '1'. Figure E-9 shows how the addition of an *else* clause completes the definition of the behavior of an *if* statement. In this description a simple 2-to-1 multiplexer is modeled using an *if* statement with an *else* clause. The synthesized and optimized result is shown in Figure E-10.

```

entity MUX21 is
    port (A, B, C : in std_logic;
          D : out std_logic);
end MUX21;

architecture RTL of MUX21 is
begin
    process (A, B, C)
    begin
        if (A = '1') then
            D <= B;
        else
            D <= C;
        end if;
    end process;
end RTL;

```

Figure E-9 Code example for *if-else* statement.

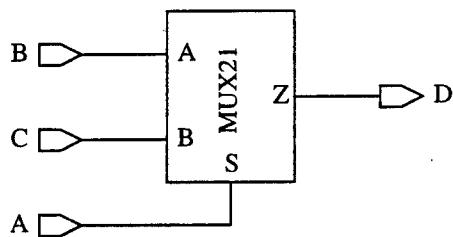


Figure E-10 Synthesized hardware for *if-else* statement.

The *case* statement controls the execution of one or more sequential statements based on the value of an *expression*. VHDL requires all possible values for a selector must be described in a *case* statement. This can be done by having as many *when* clauses as selector *choices* or by use of a *when others* clause. The *case* statement implies a multiplexing architecture. For example, consider the description in Figure E-11. The selector for the *case* statement is *SEL*. Since *SEL* is two bits wide, there are 2^2 possible selector values for synthesis purposes. The *when others* clause accounts for other values not specified with a

when clause. The gate level design is implemented by a 4-by-1 multiplexer architecture as shown in Figure E-12.

```

entity MUX4 is
    port (DATA_IN : in std_logic_vector(3 downto 0);
          SEL : in std_logic_vector(1 downto 0);
          DATA_OUT : out std_logic);
end MUX4;

architecture RTL of MUX4 is
begin
    process (SEL, DATA_IN)
    begin
        case SEL is
            when "00" => DATA_OUT <= DATA_IN(0);
            when "01" => DATA_OUT <= DATA_IN(1);
            when "10" => DATA_OUT <= DATA_IN(2);
            when others => DATA_OUT <= DATA_IN(3);
        end case;
    end process;
end RTL;

```

Figure E-11 Code example for *case* statement.

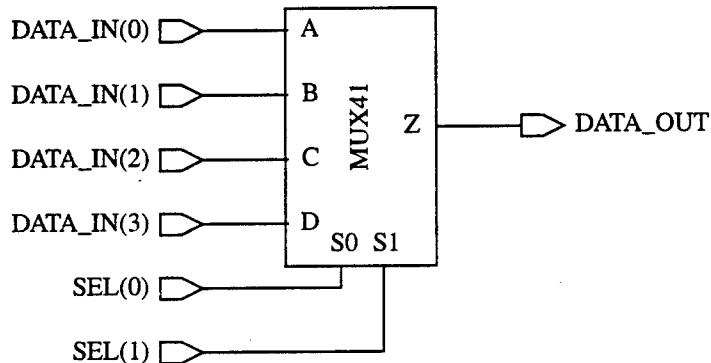


Figure E-12 Synthesized hardware for *case* statement.

AutoLogic II also supports indexed assignments using *variables*. In the following example *INDEX* is used as a *variable* index to the vector *DATA_IN*. An alternate architecture RTL2 can be used to describe the behavior of the entity MUX4 in Figure E-11. A function, *to_integer*, is assumed available to convert a 2-bit bit_vector to its integer representation.

```
architecture RTL2 of MUX4 is
begin
  process (SEL, DATA_IN)
    variable INDEX : integer;
  begin
    INDEX := to_integer(SEL);
    DATA_OUT <= DATA_IN(INDEX);
  end process;
end RTL2;
```

Figure E-13 Code example for variable index assignment.

The *signal* assignment statement

```
DATA_OUT <= DATA_IN(INDEX);
```

is equivalent to the *case* statement in Figure E-11 and results in the same multiplexer architecture in Figure E-12.

Appendix F

VHDL Source Code

This appendix contains the VHDL source code for the examples used throughout this dissertation. The examples are grouped according to the tables of fault experiment results found in Appendix B. The behavioral description is first included as **example.vhd**, followed by the behavioral test vectors in WAVES format, **example_vectors.txt**. Lastly, one or more structural descriptions are outlined for gate level implementations. The VHDL structural descriptions are numbered for multiple realizations as **example_structure1.vhd** and **example_structure2.vhd**.

F.1 CASE1**F.1.1 case1.vhd**

```

entity CASE1 is
port (Y1,Y0: in BIT; SEL: in BIT_VECTOR(1 downto 0);
      Z: out BIT);
end CASE1;

architecture BEHAVE of CASE1 is
begin
process (SEL,Y2,Y1,Y0)
begin
  case SEL is
    when "00" =>
      Z <= Y0;
    when "01" =>
      Z <= Y1;
    when others =>
      Z <= Y2;
  end case;
end process;
end BEHAVE;

```

F.1.2 case1_vectors.txt

```

$ SEL Y2 Y1 Y0 Z : time ;
$ Clause-CORRUPT(AND)
00 001 1 : 500 ns ;
01 010 1 : 500 ns ;
10 1X0 1 : 500 ns ;
11 10X 1 : 500 ns ;
$ Clause-CORRUPT(OR)
00 110 0 : 500 ns ;
01 101 0 : 500 ns ;
10 0X1 0 : 500 ns ;
11 01X 0 : 500 ns ;

```

F.1.3 case1_structure1.vhd

```

-- VHD Object: Entity "case1" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
case1(case1)")
-- Generated on: Fri Feb 13 12:10:22 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/case1/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY ieee;
-- PACKAGE STATEMENT
use ieee.std_logic_1164.all;
entity case1 is
  -- PORT LIST
  port(
    sel : in std_logic_vector(1 downto 0);
    y0 : in std_logic;
    y1 : in std_logic;
    y2 : in std_logic;
    z : out std_logic
  );
end case1;

```

F.1.4 case1_structure2.vhd

```

-- VHD Object: Architecture "structure1" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
case1(case1.structure1)") of entity "case1" (component interface "/home/rjh6v/csis/mentor/B4/eddm/
work/case1(case1.structure1)")
-- Generated on: Fri Feb 13 15:48:53 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/case1(case1.structure1)
-- Program: VHDLwrite v8.5_2.4 wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure1 of case1 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component or2
    component or3p
  end component;
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure1;

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of case1 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component and2
    component and3
  end component;
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;

```

F.2 ARRAY4**F.2.1 array4.vhd**

```
library ieee;
use ieee.std_logic_1164.all;
entity ARRAY4 is
port(Y: in BIT_VECTOR(3 downto 0);
    I: in INTEGER range 3 downto 0;
    Z: out BIT);
end ARRAY4;
```

```
architecture BEHAVE of ARRAY4 is
```

```
begin
process(Y,I)
```

```
begin
    Z <= Y(I);
end process;
```

```
end BEHAVE;
```

F.2.2 array4_vectors.txt

```
% Y I Z : time:
X110 00 0 : 500 ns;
X001 00 1 : 500 ns;
IX01 01 0 : 500 ns;
IX10 01 1 : 500 ns;
1011 10 0 : 500 ns;
0110 10 1 : 500 ns;
0111 11 0 : 500 ns;
100X 11 1 : 500 ns;
```

F.2.3 array4_structure1.vhd

```
-- VHDL object: Entity "array4" (component interface /home/rjh6v/csis/mentor/B4/eddm/work/
array4:array4)
-- Generated on: Wed Jun 10 10:00:18 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/array4/part
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
-- LIBRARY STATEMENT
```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
```

```
entity array4 is
```

```
-- PORT LIST
```

```
port(
    I : in std_logic_vector(3 downto 0);
    Y : in std_logic_vector(11 downto 0);
    Z : out std_logic );
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
array4:array4:array4:array4")
-- Generated on: Wed Jun 10 10:00:18 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/array4/array4_gen_low_s
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
library work;
use ieee.std_logic_1164.ALL;
architecture structure of array4 is
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
```

F.2.4 array4_structure2.vhd

```
component and3
    component or4;
    -- INLINE CONFIGURATIONS
begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
    end structure1;
    -- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
array4:array4:xi72a_s") of entity "array4" (component interface /home/rjh6v/csis/mentor/B4/
eddm/work/array4:array4:xi72a_s)
    -- Generated on: Wed Jul 29 15:13:44 1998
    -- Generated by: rjh6v
    -- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/array4:array4_xi72a_s
    -- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
library work;
use ieee.std_logic_1164.ALL;
architecture structure2 of array4 is
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
    component and2
        component inv
            component or2;
            -- INLINE CONFIGURATIONS
begin
            -- CONCURRENT SIGNAL ASSIGNMENTS
            end structure2;
            -- VHDL object: Entity "array4" (component interface /home/rjh6v/csis/mentor/B4/eddm/work/
array4:array4)
            -- Generated on: Wed Jun 10 10:00:18 1998
            -- Generated by: rjh6v
            -- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/array4/part
            -- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
-- LIBRARY STATEMENT
```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
```

```
entity array4 is
```

```
-- PORT LIST
```

```
port(
    I : in std_logic_vector(3 downto 0);
    Y : in std_logic_vector(11 downto 0);
    Z : out std_logic );
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
array4:array4:gen_low_s") of entity "array4" (component interface /home/rjh6v/csis/mentor/B4/
eddm/work/array4:array4)
-- Generated on: Wed Jun 10 10:00:18 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/array4:array4_gen_low_s
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
library work;
use ieee.std_logic_1164.ALL;
architecture structure of array4 is
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
```

F.3 SHIFT4u

F.3.1 shift4u.vhd

```

library IEEE; ARITHMETIC;
use IEEE.STD.LOGIC_1164.all;
use ARITHMETIC.SCD_LOGIC.ARITH.all;

entity shift4u is
port(
    OP: in STD_LOGIC_VECTOR(1 DOWNTO 0);
    A: in STD_LOGIC_VECTOR(3 DOWNTO 0);
    D: out STD_LOGIC_VECTOR(3 DOWNTO 0));
end shift4u;

architecture behave of shift4u is
begin
process(OP_A)
variable TMP: STD_LOGIC_VECTOR(3 DOWNTO 0);
begin
case OP is
when "01" =>
for I in 0 to 2 loop
    TMP(I) := A(I+1);
end loop;
TMP(3) := '0';
when "10" =>
for I in 3 DOWNTO 1 loop
    TMP(I) := A(I-1);
end loop;
TMP(0) := '0';
when others =>
    TMP := A;
end case;
D <= TMP;
end process;
end behave;
end shift4u;

```

F.3.2 shift4u_vectors.txt

```

$ OP A_D : time;
00 0101 0101 : 500 ns;
00 1010 1010 : 500 ns;
01 0101 0010 : 500 ns;
01 0101 0101 : 500 ns;
10 0101 1010 : 500 ns;
10 1010 0100 : 500 ns;
11 0101 0101 : 500 ns;
11 1010 1010 : 500 ns;

```

F.3.3 shift4u_structure1.vhd

```

-- VHDL object: Entity "shift4u" (component interface )
-- Generated on: Mon Jun  8 16:17:34 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/shift4u/part
-- Program: VHDLwrite v8.5.2-4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;

```

F.3.4 shift4u_structure2.vhd

```

-- VHDL object: Architecture "structure" (schematic )
-- Generated on: Mon Jun  8 16:17:34 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/shift4u/shift4u
-- Program: VHDLwrite v8.5.2-4 Wed Feb 5 16:56:22 PST 1997
library ieee;
use ieee.std_logic_1164.all;
architecture structure of shift4u is
begin
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
    component inv
    component nand2
    component nand3
    component or2
    component xor2
    -- INLINE CONFIGURATIONS
    begin
        -- CONCURRENT SIGNAL ASSIGNMENTS
        -- COMPONENT INSTANTIATIONS
    end structure;
end shift4u;

```

F.4 LESS2

F.4.1 less2.vhd

```
library ieee;
use ieee.std_logic_1164.all;
entity LESS2 is
port (A,B: in BIT_VECTOR(1 downto 0); LT: out BIT);
end LESS2;
architecture BEHAVE of LESS2 is
begin
process (A,B)
begin
if A < B then
  LT <= '1';
else
  LT <= '0';
end if;
end process;
end BEHAVE;
```

F.4.2 less2_vectors.txt

```
% A B LT : time;
% Class I
0 0 0 : 500 ns;
0 0 1 : 500 ns;
0 1 0 : 500 ns;
0 1 1 : 500 ns;
1 0 0 : 500 ns;
1 0 1 : 500 ns;
1 1 0 : 500 ns;
1 1 1 : 500 ns;
```

F.4.3 less2_structure1.vhd

```
-- VHDL object: Entity "less2" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/less2")
-- Generated on: Mon Aug 3 15:25:34 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/less2/part
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
use ieee.std_logic_1164.all;
entity less2 is
  port(
    a : in std_logic_vector(1 downto 0);
    b : in std_logic_vector(1 downto 0);
    lt : out std_logic );
end less2;
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/less2/less2_gen_low_s") of entity "less2"
-- Generated on: Mon Aug 3 15:25:34 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/less2/gen_low_s
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
```

```
use ieee.std_logic_1164.all;
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
  component inv
  component or2
begin
  -- INLINE CONFIGURATIONS
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure;
```

F.4.4 less2_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/less2/less2_gen_low_s") of entity "less2"
-- Generated on: Mon Aug 3 15:29:10 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/less2/less2_gen_low_s
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 EST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of less2 is
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
  component inv
  component or2
begin
  -- INLINE CONFIGURATIONS
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;
```

library ieee;

library work;

use ieee.std_logic_1164.all;

architecture structure2 of less2 is

-- SIGNAL DECLARATIONS

-- COMPONENT DECLARATIONS

component and2

component inv

component or2

begin
 -- INLINE CONFIGURATIONS
 -- CONCURRENT SIGNAL ASSIGNMENTS
 -- COMPONENT INSTANTIATIONS
end structure2;

F.5 EQUAL3

F.5.1 equal3.vhd

```
library ieee;
library work;
use ieee.std_logic_1164.all;
entity EQUAL3 is
port(A,B: in STD_VECTOR(2 downto 0); EQ: out BIT);
end EQUAL3;

architecture BEHAVE of EQUAL3 is
begin
process(A,B)
begin
if A = B then
EQ <= '1';
else
EQ <= '0';
end if;
end process;
end BEHAVE;
```

F.5.2 equal3_vectors.txt

```
% A B EQ : time;
% Class I
000 000 1 : 500 ns;
111 111 1 : 500 ns;
% Class II
000 001 0 : 500 ns;
000 010 0 : 500 ns;
000 100 0 : 500 ns;
001 000 0 : 500 ns;
010 000 0 : 500 ns;
100 000 0 : 500 ns;
```

F.5.3 equal3_structure1.vhd

```
-- VHDL object: Entity "equal3" (component interface "home/rjh6v/csis/mentor/B4/eddm/work/equal3")
-- Generated on: Mon Aug 3 15:33:29 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/equal3/part
-- Program: VHDlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity equal3 is
-- PORT LIST
port(
a : in STD_LOGIC_VECTOR(2 DOWNTO 0);
b : in STD_LOGIC_VECTOR(2 DOWNTO 0);
eq : out STD_LOGIC );
end equal3;
-- VHDL object: Architecture "structure" (schematic "home/rjh6v/csis/mentor/B4/eddm/work/equal3_low_s" of entity "equal3")
-- Generated on: Mon Aug 3 15:33:29 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/equal3/equal3_low_s
-- Program: VHDlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
```

F.5.4 equal3_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "home/rjh6v/csis/mentor/B4/eddm/work/equal3_low_s" of entity "equal3")
-- Generated on: Mon Aug 3 15:39:39 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/equal3_low_s
-- Program: VHDlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of equal3 is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component xor2
-- IN LINE CONFIGURATIONS
begin
-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
end structure2;
```

E.6 GREATER3

E.g. greater3.vhd

```

library ieee;
use ieee.std_logic_1164.all;

entity GREATER3 is
port(A,B: in INTEGER range -4 to +3;
      GT: out BIT);
end GREATER3;

architecture BEHAVE of GREATER3 is
begin
process(A,B)
begin
  if A > B then
    GT <= '1';
  else
    GT <= '0';
  end if;
end process;
end BEHAVE;

```

E.6.2 greater3 vectors.txt

卷之三

```
-- VHDLObject: Entity "greater3" (component interface) '/home/rjh6v/csis/mentor/B4/eddm/work/greater3:greater3'
-- Generated on: Tue May 26 14:42:29 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/greater3/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned;
USE ieee.std_logic_arith;
```

F.6.5 greater3_structure3.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/greater3/greater3_sub_s").ofanty.greater3*(componentinterface"/home/rjh6v/csis/mentor/B4/eddm/work/greater3_sub_s");
-- Generated on: Thu Nov 5 14:32:34 1998
-- Generated by: rjh6v /home/rjh6v/csis/mentor/B4/eddm/work/greater3/greater3_sub_s
-- Source from: VHDlwrite v8.5.2.4 Wed Feb 5 10:56:22 PST 1997
-- Program: VHDlwrite

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure3 of greater3 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
    component and3
    component inv
    component or2
    component xor2
  end component;
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure3;
```

F.7 LE5

F.7.1 le5.vhd

```
library ieee;
use ieee.std_logic_1164.all;
entity LE5 is
port(A: in INTEGER range -16 to +15;
     LE: out BIT);
end LE5;

architecture BEHAVE of LE5 is
begin
process(A)
begin
  if A <= 5 then
    LE <= '1';
  else
    LE <= '0';
  end if;
end process;
end BEHAVE;
```

F.7.2 le5_vectors.txt

```
% A : LE : time:
11111 1 : 500 ns;
00011 1 : 500 ns;
00101 1 : 500 ns;
00110 0 : 500 ns;
01000 0 : 500 ns;
01100 0 : 500 ns;
```

F.7.3 le5_structure1.vhd

```
-- VHDL object: Entity "le5" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/le5:le5")
-- Generated on: Fri May 29 14:29:46 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/le5/part
-- Program: VHDWriter v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-- PORT LIST
port(
  a : in std_logic_vector(4 downto 0);
  le : out std_logic
);
end le5;

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/le5:le5_gen_meds") of entity "le5"
-- Generated on: Fri May 29 14:29:46 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/le5_gen_meds
-- Program: VHDWriter v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
```

F.7.4 le5_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/le5:le5_xi72a_s") of entity "le5" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/le5:le5_xi72a_s")
-- Generated on: Fri Jul 24 13:48:28 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/le5_xi72a_s
-- Program: VHDWriter v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of le5 is
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;
```

F.8 GE23u

F.8.1 ge23u.vhd

```
library ieee;
use ieee.std_logic_1164.all;
entity GE23u is
port (A: in INTEGER range 0 to 31;
      GE: out BIT);
end GE23u;
```

architecture BEHAVE of GE23u is

begin

process(A)

begin

if A >= 23 then

 GE <= '1';

else

 GE <= '0';

end if;

end process;

F.8.2 ge23u_vectors.txt

```
% A   GE : time;
01111 0 : 500 ns;
10001 0 : 500 ns;
10101 0 : 500 ns;
10110 0 : 500 ns;
10111 1 : 500 ns;
11000 1 : 500 ns;
11010 1 : 500 ns;
11110 1 : 500 ns;
```

F.8.3 ge23u_structure1.vhd

```
-- VHDL object: Entity' ge23u' (component interface "/home/rjh6v/csis/mentor/B4 /eddm/work/
ge23u/ge23u")
-- Generated on: Fri May 29 14:15:26 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4 /eddm/work/ge23u/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

-- LIBRARY STATEMENT

```
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.ALL;
entity ge23u is
```

-- PORT LIST

```
port
  a : in std_logic_vector(4 downto 0);
end ge23u;
```

```
-- VHDL object: Architecture' structure" (schematic "/home/rjh6v/csis/mentor/B4 /eddm/work/
ge23u/ge23u.gen.med_s")of entity'ge23u'(componentinterface"/home/rjh6v/csis/mentor/B4 /eddm/
work/ge23u/ge23u")
-- Generated on: Fri May 29 14:15:26 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4 /eddm/work/ge23u/gen_med_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
-- LIBRARY IEEE;
library work;
```

```
use ieee.std_logic_1164.ALL;
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
  component or2
    begin
      -- INLINE CONFIGURATIONS
      -- CONCURRENT SIGNAL ASSIGNMENTS
      -- COMPONENT INSTANTIATIONS
    end structure1;
```

F.8.4 ge23u_structure2.vhd

```
-- VHDL object: Architecture' structure" (schematic "/home/rjh6v/csis/mentor/B4 /eddm/work/
ge23u/ge23u.xi72a.s")of entity'ge23u'(componentinterface"/home/rjh6v/csis/mentor/B4 /eddm/
work/ge23u/ge23u")
-- Generated on: Fri Jul 24 13:44:22 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4 /eddm/work/ge23u/xi72a.s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

library ieee;

library work;

```
use ieee.std_logic_1164.ALL;
architecture structure2 of ge23u is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
-- component and2
  component or2
    begin
      -- INLINE CONFIGURATIONS
      -- CONCURRENT SIGNAL ASSIGNMENTS
      -- COMPONENT INSTANTIATIONS
    end structure2;
```

F.9 LT12u**F.9.1 LT12u.vhd**

```

library ieee;
use ieee.std_logic_1164.all;
entity Lt12u is
port(A: in INTEGER range 0 to 31;
      LT: out BIT);
end Lt12u;

architecture BEHAVE of Lt12u is
begin
process(A)
begin
  if A < 12 then
    LT <= '1';
  else
    LT <= '0';
  end if;
end process;
end BEHAVE;

```

F.9.2 Lt12u_vectors.txt

```

$ A LT : time;
00111 1 : 500 ns;
01011 1 : 500 ns;
01100 0 : 500 ns;
10000 0 : 500 ns;

```

F.9.3 Lt12u_structure.vhd

```

-- VHDL object: Entity "lt12u" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
lt12u:lt12u")
-- Generated on: Fri May 29 14:11:06 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/lt12u/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity lt12u is
-- PORT LIST
port
  a : in std_logic_vector(4 downto 0);
  lt : out std_logic
);
end lt12u;
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
lt12u/lt12u.gen.med_s")of entity "lt12u"(componentinterface"/home/rjh6v/csis/mentor/B4/eddm/
work/lt12u:lt12u")
-- Generated on: Fri May 29 14:11:06 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/lt12u/lt12u_gen_med_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;

```

F.10 GT3n

F.10.1 gt3n.vhd

```

library ieee;
use ieee.std_logic_1164.all;
architecture BEHAVE of gt3n is
begin
  process(A)
    begin
      if A > -3 then
        GT <= '1';
      else
        GT <= '0';
      end if;
    end process;
  end BEHAVE;

```

F.10.2 gt3n_vectors.txt

```

% A  GT : time:
10111 0 : 500 ns;
11011 0 : 500 ns;
11010 0 : 500 ns;
11110 1 : 500 ns;
00000 1 : 500 ns;

```

F.10.3 gt3n_structure.vhd

```

-- VHDL object: Entity "gt3n" (component interface
--   generated on: Fri May 29 14:24:50 1998
--   Generated by: rjh6v
--   Source from: /home/rjh6v/csis/mentor/B4/eddm/work/gt3n/part
--   Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY IEEE;
-- PACKAGE STATEMENT
use ieee.std_logic_1164.all;
entity gt3n is
  port(
    a : in std_logic_vector(4 downto 0);
    gt : out std_logic
  );
end gt3n;

-- VHDL object: Architecture "structure" (schematic
--   generated on: Fri May 29 14:24:50 1998
--   Generated by: rjh6v
--   Source from: /home/rjh6v/csis/mentor/B4/eddm/work/gt3n_gen_med_s
--   Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

F.11 COMPARE

F.11.1 compare.vhd

```

library ieee;
use ieee.std_logic_1164.all;

entity COMPARE is
    port(A, B: in BIT_VECTOR(1 downto 0); Y2, Y1, Y0: out BIT);
end COMPARE;

architecture BEHAVE of COMPARE is
begin
process(A,B,Y2,Y1,Y0)
begin
    if A > B then
        Z <= Y2;
    elsif A < B then
        Z <= Y0;
    else
        Z <= Y1;
    end if;
end process;
end BEHAVE;

```

F.11.4 compare_structure1.vhd

```

-- VHDL object: Entity "compare" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
compare:compare")
-- Generated on: Sat Jan 31 11:48:58 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/compare/part
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
library ieee;
-- PACKAGE STATEMENT
use ieee.std_logic_1164.all;
entity compare is
    -- PORT LIST
    port(
        a : in std_logic_vector(1 downto 0);
        b : in std_logic_vector(1 downto 0);
        y0 : in std_logic;
        y1 : in std_logic;
        y2 : in std_logic;
        z : out std_logic
    );
end compare;

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
compare/compare_low_s") of entity "compare" (ComponentInterface "/home/rjh6v/csis/mentor/B4/
eddm/work/compare:compare")
-- Generated on: Sat Jan 31 11:48:58 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/compare/compare_low_s
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of compare is
    -- SIGNAL DECLARATIONS
    -- TYPE DECLARATIONS
    -- COMPONENT DECLARATIONS
    component and2
        component inv
        component invr
        component nand2
        component nor2p
            component or2
                -- INLINE CONFIGURATIONS
            begin
                -- CONCURRENT SIGNAL ASSIGNMENTS
                -- COMPONENT INSTANTIATIONS
            end structure;

```

F.11.2 compare_vectors.txt

```

% A B Y2 Y1 Y0 Z : time ;
% Group I
0000 100 0 : 500 ns ;
0101 010 1 : 500 ns ;
0101 100 1 : 500 ns ;
1010 101 0 : 500 ns ;
1111 010 1 : 500 ns ;
% Group II
0100 100 1 : 500 ns ;
1001 011 0 : 500 ns ;
1001 100 1 : 500 ns ;
1110 011 0 : 500 ns ;
% Group III
0001 001 1 : 500 ns ;
0110 110 0 : 500 ns ;
0110 001 1 : 500 ns ;
1011 110 0 : 500 ns ;
% Group IV
0001 110 0 : 500 ns ;
0110 001 1 : 500 ns ;
0110 110 0 : 500 ns ;
1011 001 1 : 500 ns ;

```

F.11.3 compare_vectors.alt

```

% A B Y2 Y1 Y0 Z : time ;
% Group I
0000 010 1 : 500 ns ;
0101 101 0 : 500 ns ;
1010 010 1 : 500 ns ;
1111 101 0 : 500 ns ;
% Group II
0100 011 0 : 500 ns ;
1001 100 1 : 500 ns ;
1001 011 0 : 500 ns ;
1110 100 1 : 500 ns ;
% Group III
0001 110 0 : 500 ns ;
0110 001 1 : 500 ns ;
0110 110 0 : 500 ns ;
1011 001 1 : 500 ns ;

```

F.11.5 compare_structure2.vhd

```
-- VHDL object: Architecture "structure2" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/compar
-- compare/compare_med_s") of entity "compare" (componentinterface "/home/rjh6v/csis/mentor/B4/
-- Generated on: Sat Jan 31 11:56:27 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/compare/compare_med_s
-- Program: VHDIwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of compare is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
    component invr
    component nand2
    component or2
  begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
  end structure2;
```

F.12 COMPARE3

F.12.1 compare3.vhd

```

library ieee;
entity COMPARE3 is
port(A,B: in BIT_VECTOR(2 downto 0); Y2,Y1,Y0: in BIT; Z: out BIT);
end COMPARE3;

architecture BEHAVE of COMPARE3 is
begin
process(A,B,Y2,Y1,Y0)
begin
if A > B then
  Z <= Y2;
elsif A < B then
  Z <= Y0;
else
  Z <= Y1;
end if;
end process;
end BEHAVE;

```

F.12.2 compare3_vectors.txt

```

% A B Y2 Y1 Y0 Z : time ;
% Group I
000 000 100 0 : 500 ns ;
001 001 100 1 : 500 ns ;
011 010 011 0 : 500 ns ;
100 011 011 0 : 500 ns ;
101 100 100 1 : 500 ns ;
110 101 100 1 : 500 ns ;
111 110 011 0 : 500 ns ;
000 000 100 1 : 500 ns ;
001 001 001 1 : 500 ns ;
010 010 110 0 : 500 ns ;
011 100 110 0 : 500 ns ;
100 101 001 1 : 500 ns ;
110 110 001 1 : 500 ns ;
111 111 110 0 : 500 ns ;

-- VHDL object: Entity "compare3" (component interface) "/home/rjh6v/cssis/mentor/B4/eddm/work/
compare3;compare3"
-- Generated on: Mon Feb 9 10:39:48 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/cssis/mentor/B4/eddm/work/compare3/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity compare3_is
port(
  a : in std_logic_vector(2 downto 0);
  b : in std_logic;
  y0 : in std_logic;
  y1 : in std_logic;
  y2 : in std_logic;
  z : out std_logic
);
end compare3;
-- VHDL object: Architecture "structure1" (schematic) "/home/rjh6v/cssis/mentor/B4/eddm/work/
compare3/compare3_low's'identity compare3" (component interface) "/home/rjh6v/cssis/mentor/B4/
eddm/work/compare3;compare3"
-- Generated on: Mon Feb 9 10:39:48 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/cssis/mentor/B4/eddm/work/compare3/compare3_low_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of compare3 is
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
component and3
component invr
component nand2
component nand3
component or2
-- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure;

```

F.12.3 compare3_vectors.alt

```

% A B Y2 Y1 Y0 Z : time ;
% Group I
000 000 010 1 : 500 ns ;
001 001 010 1 : 500 ns ;
010 010 010 0 : 500 ns ;
011 011 101 0 : 500 ns ;
100 101 010 1 : 500 ns ;
101 110 110 0 : 500 ns ;
110 111 001 1 : 500 ns ;
111 111 101 0 : 500 ns ;

```

F.12.5 compare3_structure2.vhd

```
-- VHDL object: Architecture "structure2" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/compare3/compare3_med_s") of entity "compare3" (componentInterface "/home/rjh6v/csis/mentor/B4/eddm/work/compare3/compare3")
-- Generated on: Mon Feb 9 10:47:58 1998
-- Generated by: rjdev /home/rjh6v/csis/mentor/B4/eddm/work/compare3/compare3_med_s
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/compare3/compare3_med_s
-- Program: VHDIwrite v8.5.2.4 Wed Feb 5 10:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of compare3 is
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
  component and3
  component inv
  component nand2
  component nand3
  component nor2p
  component or2
  component or3p
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;
```

F.13 COMPARE4

F.13.1 compare4.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity COMPARE4 is
port(A,B: in BIT_VECTOR(1 downto 0); Y2,Y1,Y0: in BIT_VECTOR(3 downto 0);
2: out BIT_VECTOR(3 downto 0));
end COMPARE4;

architecture BEHAVE of COMPARE4 is
begin
process(A,B,Y2,Y1,Y0)
begin
if A > B then
elsif A < B then
else
2 <= Y1;
end if;
end process;
end BEHAVE;
```

F.13.4 compare4_structure.vhd

```
-- VHDL object: Entity "compare4" (componentinterface "/home/rjh6v/csis/mentor/B4/eddm/work/
compare:compare4")
-- Generated on: Sat Jan 31 16:05:59 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/compare4/part
-- Program: VHDWriter v8.5-2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity compare4 is
-- PORT LIST
port(
a : in std_logic_vector(1 downto 0);
b : in std_logic_vector(1 downto 0);
y0 : in std_logic_vector(3 downto 0);
y1 : in std_logic_vector(3 downto 0);
y2 : in std_logic_vector(3 downto 0);
z : out std_logic_vector(3 downto 0)
);
end compare4;
```

F.13.2 compare4_vectors.txt

```
% A B Y2 Y1 Y0 Z : time ;
% Group I
0000 1111 0000 1111 0000 : 500 ns ;
0101 1111 0000 1111 1111 : 500 ns ;
1010 1111 0000 1111 0000 : 500 ns ;
1010 1111 0000 1111 0000 : 500 ns ;
% Group II
0100 1111 0000 0000 1111 : 500 ns ;
1001 1111 0000 0000 1111 : 500 ns ;
1001 1111 0000 0000 1111 : 500 ns ;
1110 0000 1111 1111 0000 : 500 ns ;
% Group III
0001 0000 0000 1111 1111 : 500 ns ;
0110 1111 0000 0000 0000 : 500 ns ;
0110 0000 0000 1111 1111 : 500 ns ;
1011 1111 1111 0000 0000 : 500 ns ;
```

F.13.3 compare4_vectors.alt

```
% A B Y2 Y1 Y0 Z : time ;
% Group I
0000 0000 1111 0000 1111 : 500 ns ;
0101 1111 0000 1111 0000 : 500 ns ;
1010 0000 1111 0000 1111 : 500 ns ;
1111 1111 0000 1111 0000 : 500 ns ;
% Group II
0100 0000 1111 1111 0000 : 500 ns ;
1001 1111 0000 0000 1111 : 500 ns ;
1001 0000 1111 1111 0000 : 500 ns ;
1110 1111 0000 0000 1111 : 500 ns ;
% Group III
0001 1111 1111 0000 0000 : 500 ns ;
0110 0000 0000 1111 1111 : 500 ns ;
0110 1111 1111 0000 0000 : 500 ns ;
1011 0000 0000 1111 1111 : 500 ns ;
```


F.15 ADD4

F.15.1 add4.vhd

```

library IEEE; ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity add4 is
port( A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      D: out std_logic_vector(3 downto 0)
    );
end add4;

```

F.15.2 add4_vectors.txt

```

architecture behave of add4 is
begin
process(A, B)
begin
D <= A + B;
end process;
end behave;

```

```

$ A B D : time ;
0000 0000 0000 : 500 ns;
0000 1111 1111 : 500 ns;
1111 0000 1111 : 500 ns;
1111 1111 1111 : 500 ns;
$ 0001 1111 0000 : 500 ns;
1111 0001 0000 : 500 ns;
0101 0101 1010 : 500 ns;
1010 1010 0100 : 500 ns;
$
```

F.15.3 add4_vectors.clá

```

$ A B D : time ;
0000 0000 0000 : 500 ns;
0000 1111 1111 : 500 ns;
1111 0000 1111 : 500 ns;
1111 1111 1110 : 500 ns;
$ 0001 1111 0000 : 500 ns;
1111 0001 0000 : 500 ns;
0101 0101 1010 : 500 ns;
1010 1010 0100 : 500 ns;
$ 0010 1111 0001 : 500 ns;
0001 1101 1110 : 500 ns;
0001 1011 1100 : 500 ns;
```

F.15.4 add4_structure1.vhd

```

-- VHDL object: Entity "add4" (component interface *)/home/rjh6v/csis/mentor/B4/eddm/work/
add: add4;
-- Generated on: Thu Feb 26 08:52:15 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add4/part
-- Program: vhdlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

```

library ieee;
use ieee.std_logic_1164.all;
architecture structure1 of add4 is
begin
  -- PORT LIST
  port( a : in std_logic_vector(3 downto 0);
        b : in std_logic_vector(3 downto 0);
        d : out std_logic_vector(3 downto 0)
      );
end add4;

```

```

-- VHDL object: Architecture "structure" (schematic *)/home/rjh6v/csis/mentor/B4/eddm/work/
add4_low_s":ofentity"add4"(componentinterface*)/home/rjh6v/csis/mentor/B4/eddm/work/
add: add4;
-- Generated on: Thu Feb 26 08:52:45 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add4/add4_low_s
-- Program: vhdlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

```

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of add4 is
begin
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
    component buf
    component or3
    component xor2
  begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
    end structure1;

```

```

-- VHDL object: Architecture "structure" (schematic *)/home/rjh6v/csis/mentor/B4/eddm/work/
add4_cla:schematic":entity"add4_cla"(component interface *)/home/rjh6v/csis/mentor/B4/eddm/
scheme/add4_cla:add4_cla;
-- Generated on: Thu Feb 26 09:35:32 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/schem/add4_cla/schematic
-- Program: vhdlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

```

library ieee;
use ieee.std_logic_1164.all;
architecture structure2 of add4 is
begin
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component hand2
    component hand3
  begin
    -- INLINE CONFIGURATIONS
    end structure2;

```

```

-- VHDL object: Entity "add4" (component interface *)/home/rjh6v/csis/mentor/B4/eddm/work/
add: add4;
-- Generated on: Thu Feb 26 08:52:15 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add4/part
-- Program: vhdlwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

F.15.6 add4_structure3.vhd

```
-- VHDL object: Architecture "structure" (schematic) of entity "add4_nor" (component interface)
-- scheme/add4_nor: add4_nor
-- Generated on: Thu Feb 26 09:56:16 1998
-- Generated by: rjhav
-- Source from: /home/rjhav/csis/mentor/B4/schem/add4_nor/schematic
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 10:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure3 of add4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component nor2
      component nor3
        component nor4
          begin
            -- COMPONENT CONFIGURATIONS
            -- INSTANTIATIONS
          end
        end
      end
    end
  end
end structure3;
```

F.16 ADD4wc

F.16.1 add4wc.vhd

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity add4wc is
port(
    A: in std_logic_vector(3 downto 0);
    B: in std_logic_vector(3 downto 0);
    D: out std_logic_vector(3 downto 0);
    CIN: in std_logic;
    COUT: out std_logic
);
end add4wc;

architecture behave of add4wc is
begin
process(A,B,CIN)
variable operand1, operand2 : std_logic_vector(4 downto 0);
variable sum : std_logic_vector(4 downto 0);
variable carry_in : std_logic_vector(1 downto 0);
begin

```

```
    end process;  
end behave;
```

```

A B CIN COUT D : time ;
0000 0000 0 00000 : 500 ns;
0000 1111 0 11111 : 500 ns;
1111 0000 0 00001 : 500 ns;
1111 1111 1 11111 : 500 ns;

A B CIN COUT D : time ;
0000 1111 1 00000 : 500 ns;
1111 0000 1 00001 : 500 ns;
0101 0101 0 10101 : 500 ns;
1010 1010 1 01011 : 500 ns;

§ F.16.3 add4wc_vectors.cia

```

E16.3 add4vec vectors.cla

```

-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
begin
    component and2
    component buf
    component or3
    component xor2

    -- INLINE CONFIGURATIONS
    begin
        -- CONCURRENT SIGNAL ASSIGNMENTS
        -- COMPONENT INSTANTIATIONS
        end;

```

F.16.5 add4wc_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/scheme/add4wc_cla/schematic") of entity "add4wc_cla"
-- Generated on: Wed Mar 4 15:04:54 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/scheme/add4wc_cla/schematic
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of add4wc is
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
  component nand2
  component nand3
  component nand4
  component nands
  -- INLINE CONFIGURATIONS
  begin
    -- COMPONENT INSTANTIATIONS
    end structure2;
```

F.16.6 add4wc_structure3.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/add4wc/add4wc_xi72a_s") of entity add4wc (component interface /home/rjh6v/csis/mentor/B4/eddm/work/add4wc)
-- Generated on: Tue Jun 23 12:56:11 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add4wc/add4wc_xi72a_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure3 of add4wc is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
  component inv
  component or2
  -- INLINE CONFIGURATIONS
  begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    end structure3;
```

F.17 ADD8

E.17.1 add8.vhd

```

library IEEE, ARITHMETIC;
use IEEE.scd_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity add8 is
port(
    A: in std_logic_vector(7 downto 0);
    B: in std_logic_vector(7 downto 0);
    D: out std_logic_vector(7 downto 0)
);
end add8;

architecture behave of add8 is
begin
process (A,B)
begin
    D <= A + B;
end process;

```

E 17.2 add8 vectors.vf

```

% A B D : time ;
00000000 00000000 00000000 : 500 ns ;
00000000 00000000 11111111 : 500 ns ;

```

E.17.3 add8 vectors.cla

```

8 A B D : time;
00000000 00000000 00000000 00000000 : 500 ns;
00000000 11111111 11111111 11111111 : 500 ns;
11111111 00000000 00000000 00000000 : 500 ns;
11111111 11111111 11111111 11111111 : 500 ns;

8
00000001 11111111 00000000 00000000 : 500 ns;
11111111 00000001 00000000 00000000 : 500 ns;
01010101 01010101 10101010 01010100 : 500 ns;
10101010 10101010 01010100 10101010 : 500 ns;

8
00000010 11111111 00000001 00000001 : 500 ns;
00000010 11111111 00000011 00000011 : 500 ns;
00000010 11111111 00000111 00000111 : 500 ns;
00010000 11111111 00001111 00001111 : 500 ns;
00100000 11111111 00011111 00011111 : 500 ns;

8
00000001 11111111 11111111 11111111 : 500 ns;
00000001 11011111 11011111 11011111 : 500 ns;

8
00000010 11111101 11111101 11111101 : 500 ns;

```

E 17.5 add8 structure1 whd

```
-- VHDL object: Entity "add8" (component interface *) /home/rjh6v/csis/mentor/B4/eddm/work/
-- add8.add8
-- Generated on: Wed Mar 4 11:05:30 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add8/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:55:22 PST 1997

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
-- ENTITY STATEMENT
```

F.17.4 add8_vectors.opt

```

% A B D : time;
00000000 00000000 00000000 : 500 ns;
00000000 11111111 11111111 : 500 ns;
11111111 00000000 11111111 : 500 ns;
11111111 11111111 11111110 : 500 ns;
%
```

```

-- PORT LIST
port(
    a : in std_logic_vector(7 downto 0);
    b : in std_logic_vector(7 downto 0);
    d : out std_logic_vector(7 downto 0)
);
end add8;

-----vhdl object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
add8/add8_low_s") of entity add8 (componentinterface "/home/rjh6v/csis/mentor/B4/eddm/work/
add8;add8")
-- Generated on: Wed Mar 4 11:05:30 1998
-- Generated by: rjh6v
-- Source From: /home/rjh6v/csis/mentor/B4/eddm/work/add8/add8_low_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-----vhdl ieees:
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure1 of add8 is
    -- TYPE DECLARATIONS
    -- SIGNAL DECLARATIONS
    -- COMPONENT DECLARATIONS
    component and2
        port( buf
            component or2
                port( x02
                    -- INLINE CONFIGURATIONS
begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
end structure1;

```

F.17.6 add8_structure2.vhd

```

-----vhdl object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/schem/
add8_cla/add8_cla") of entity "add8_cla" (component interface "/home/rjh6v/csis/mentor/B4/
schem/add8_cla:add8_cla")
-- Generated on: Wed Mar 4 12:04:00 1998
-- Generated by: rjh6v
-- Source From: /home/rjh6v/csis/mentor/B4/schem/add8_cla/schematic
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-----vhdl ieees:
library work;
use ieee.std_logic_1164.all;
architecture structure2 of add8 is
    -- TYPE DECLARATIONS
    -- SIGNAL DECLARATIONS
    -- COMPONENT DECLARATIONS
    component inv
        component nand2
            component nand3
                component nand4
                    -- INLINE CONFIGURATIONS
                    begin
                        -- COMPONENT INSTANTIATIONS
                    end structure2;

```

F.18 SUB4**F.18.1 sub4.vhd**

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity sub4 is
port( A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      D: out std_logic_vector(3 downto 0)
    );
end sub4;

architecture behave of sub4 is
begin
process (A,B)
begin
  D <= A - B;
end process;
end behave;

```

F.18.2 sub4_vectors.txt

```

% A B D : time : 
0000 0000 0000 : 500 ns;
0000 1111 0001 : 500 ns;
1111 0000 1111 : 500 ns;
1111 1111 0000 : 500 ns;
%
0000 0001 1111 : 500 ns;
1110 1111 1111 : 500 ns;
0101 1010 1011 : 500 ns;
1010 0101 0101 : 500 ns;
%
0000 0000 0011 : 500 ns;
1110 1111 1111 : 500 ns;
0101 1010 1011 : 500 ns;
1010 0101 0101 : 500 ns;
%
0001 0000 0001 : 500 ns;
0010 0000 0010 : 500 ns;
%
0000 0001 1111 : 500 ns;
0000 0010 1110 : 500 ns;
0000 0100 1100 : 500 ns;
%
0001 0010 1111 : 500 ns;
0001 0100 1101 : 500 ns;
0010 0100 1110 : 500 ns;

```

F.18.3 sub4_vectors.cla

```

% A B D : time : 
0000 0000 0000 : 500 ns;
0000 1111 0001 : 500 ns;
1111 0000 1111 : 500 ns;
1111 1111 0000 : 500 ns;
%
0000 0001 1111 : 500 ns;
1110 1111 1111 : 500 ns;
0101 1010 1011 : 500 ns;
1010 0101 0101 : 500 ns;
%
0001 0000 0001 : 500 ns;
0010 0000 0010 : 500 ns;
%
0000 0001 1111 : 500 ns;
0000 0010 1110 : 500 ns;
0000 0100 1100 : 500 ns;
%
0001 0010 1111 : 500 ns;
0001 0100 1101 : 500 ns;
0010 0100 1110 : 500 ns;

```

F.18.4 sub4_structure1.vnd

```

-- VHDL object: Entity "sub4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
sub4:sub4")
-- Generated on: Thu Apr 2 11:55:41 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sub4/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
-- PORT LIST
entity sub4 is
port(
  a : in std_logic_vector(3 downto 0);
  b : in std_logic_vector(3 downto 0);
  d : out std_logic_vector(3 downto 0)
);
end sub4;

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
sub4/sub4:sub4")
-- Generated on: Thu Apr 2 11:55:41 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sub4/sub4
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of sub4 is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
component buf
component inv
component inrv
component nand2p
component or3
component xor2
-- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure;

```

F.18.5 sub4_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic */home/rjh6v/cs1s/mentor/B4/scheme/
sub4_cla/schematic*) of entity "sub4_cla" (Component interface */home/rjh6v/cs1s/mentor/B4/
scheme/sub4_cla/sub4_cla)
-- Generated on: Thu Apr 2 15:31:30 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/cs1s/mentor/B4/scheme/sub4_cla/schematic
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-----
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of sub4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
    component inv
      component nand2
        component nand3
        component nand4
      end component;
    begin
      -- INLINE CONFIGURATIONS
      -- COMPONENT INSTANTIATIONS
    end structure2;
```

F.19 ADDSUB4**F.19.1 addsub4.vhd**

```

library IEEE; ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity addsub4 is
port(
    OP: in std_logic;
    A: in std_logic_vector(3 downto 0);
    B: in std_logic_vector(3 downto 0);
    D: out std_logic_vector(3 downto 0)
);
end addsub4;

```

```

architecture behave of addsub4 is
begin
process(OP,A,B)
begin

```

```

    if OP = '0' then
        D <= A + B;
    else
        D <= A - B;
    end if;
end process;
end behave;

```

F.19.2 addsub4_vectors.txt

```

OP A B D : time ;
$ ADD
0 0000 0000 0000 : 500 ns;
0 0000 1111 1111 : 500 ns;
0 1111 0000 1111 : 500 ns;
0 1111 1111 0000 : 500 ns;
$ SUB
0 0000 0000 0000 : 500 ns;
1 0000 1111 0001 : 500 ns;
1 1111 0000 1111 : 500 ns;
1 1111 1111 0000 : 500 ns;
$ ADD CLA
0 0000 0001 1111 : 500 ns;
1 1111 1111 1111 : 500 ns;
1 0101 1010 1011 : 500 ns;
1 1010 0101 0101 : 500 ns;
$ SUB CLA
0 0001 0000 0001 : 500 ns;
1 0010 0000 0010 : 500 ns;
0 0001 0101 0110 : 500 ns;
0 0010 0110 1000 : 500 ns;
$ ADD CLA
0 0001 0011 0100 : 500 ns;
1 0010 0000 0010 : 500 ns;
0 0001 0101 0110 : 500 ns;
0 0010 0110 1000 : 500 ns;
$ SUB CLA
0 0000 0001 1111 : 500 ns;
1 0000 0010 1100 : 500 ns;
0 0000 0100 1100 : 500 ns;
1 0000 0101 1111 : 500 ns;
1 0001 0100 1101 : 500 ns;
1 0001 0101 1101 : 500 ns;

```

F.19.4 addsub4_structure1.vhd

```

-- VHDL object: Entity "addsub4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
add_sub_4;add_sub_4")
-- Generated on: Thu Apr 2 12:27:01 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add_sub_4/part
-- Program: VHDLDwrite v8.5_2.4 Wed Feb 5 16:56:22 PST-1997
-- LIBRARY IEEE;
-- PACKAGE STATEMENT
USE IEEE.STD.LOGIC.all;
entity addsub4 IS
    PORT LIST
    port(
        op : in STD_LOGIC;
        a : in STD_LOGIC_VECTOR(3 DOWNTO 0);
        b : in STD_LOGIC_VECTOR(3 DOWNTO 0);
        d : out STD_LOGIC_VECTOR(3 DOWNTO 0)
    );
end addsub4;

```

```

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
add_sub_4/add_sub_4.med.s" of entity "add_sub_4" (component interface "/home/rjh6v/csis/mentor/
B4/eddm/work/add_sub_4;add_sub_4"))
-- Generated on: Thu Apr 2 12:27:01 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/add_sub_4/add_sub_4_med_s
-- Program: VHDLDwrite v8.5_2.4 Wed Feb 5 16:56:22 PST-1997

```

F.19.3 addsub4_vectors.cla

```

OP A B D : time ;
$ ADD
0 0000 0000 0000 : 500 ns;
0 0000 1111 1111 : 500 ns;
1 1110 1111 1111 : 500 ns;
1 0101 1010 1011 : 500 ns;
1 1010 0101 0101 : 500 ns;
$ SUB
0 0000 0000 0000 : 500 ns;
1 1111 1111 0000 : 500 ns;
0 0000 1111 1111 : 500 ns;
0 1111 0000 1111 : 500 ns;
0 1111 1111 1110 : 500 ns;

```

```

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of addsub4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
    component buf
    component inv
    component or2
    component or3
    component xor2
  end component;
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure1;

```

F.19.5 addsub4_structure2.vhd

```

----- Object: Architecture "structure" of addsub4_c1a -----
-- VHDL object: Architecture "structure" of addsub4_c1a" (componentInterface/B4/schem/
addsub4_c1a/schematic")of entity"addsub4_c1a" (mentor/B4/schem/
B4/schem/addsub4_c1a:addsub4_c1a")
-- Generated on: Thu Apr 2 14:56:03 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/mentor/B4/schem/addsub4_c1a/schematic
-- Program: VHDLwrite '98.5.2.4 Wed Feb 5 16:56:22 EST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of addsub4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
  component nand2
  component nand3
  component nand4
  -- INLINE CONFIGURATIONS
begin
  -- COMPONENT INSTANTIATIONS
end structure2;

```

F.20 INC4**F.20.1 inc4.vhd**

```
library IEEE; ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity inc4 is
  port(
    Y: in INTEGER range -8 to +7;
    Z: out INTEGER range -7 to +7
  );
end inc4;
```

architecture behave of inc4 is

```
process(Y)
begin
  Z <= Y + 1;
end process;
end behave;
```

F.20.2 inc4_vectors.txt

```
$ Y .Z : time;
0000 0001 : 500 ns;
1111 0000 : 500 ns;
$ 0000 0000 : 500 ns;
0111 1000 : 500 ns;
1011 1100 : 500 ns;
1101 1110 : 500 ns;
1110 1111 : 500 ns;
```

F.20.3 inc4_structure1.vhd

```
-- VHDL object: Entity "inc4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/inc4")
-- Generated on: Tue May 26 16:18:51 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/inc4/part
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity inc4 is
  -- PORT LIST
  port(
    Y : in std_logic_vector(3 downto 0);
    Z : out std_logic_vector(3 downto 0) );
end inc4;
-- VHDL object: Architecture "structure" (schematic "inc4" (component-interface "/home/rjh6v/csis/mentor/B4/eddm/work/inc4"))
-- Generated on: Tue May 26 16:18:51 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/inc4_gen_low_s
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
```

F.20.4 inc4_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "inc4" (component-interface "/home/rjh6v/csis/mentor/B4/eddm/work/inc4"))
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
component inv
component xor2
-- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure;
```

```
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of inc4 is
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
  component inv
  component xor2
  -- INLINE CONFIGURATIONS
  begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
  end structure2;
```

F.21 INC8**F.21.1 inc8.vhd**

```

library IEEE; ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity inc8 is
  port( Y: in INTEGER range -128 to +126;
        Z: out INTEGER range -127 to +127 );
end inc8;

architecture behave of inc8 is
begin
  process(Y)
  begin
    Z <= Y + 1;
  end process;
end behave;

```

F.21.2 inc8_vectors.txt

```

$ Y Z : time;
00000000 00000001 : 500 ns;
11111111 00000000 : 500 ns;
$ 01111111 10000000 : 500 ns;
10111111 11000000 : 500 ns;
11011111 11100000 : 500 ns;
11101111 11110000 : 500 ns;
11110111 11111000 : 500 ns;
11111011 11111100 : 500 ns;
11111101 11111110 : 500 ns;
11111110 11111111 : 500 ns;

```

F.21.3 inc8_structure.vhd

```

-- VHDL object: Entity "inc8" (component interface * /home/rjh6v/csis/mentor/B4/eddm/work/
inc8")
-- Generated on: Wed Jun 3 10:53:06 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/inc8/part
-- Program: VHDWriter v8.5.2.4 Wed Feb 5 16:36:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.ALL;
entity inc8 is
  -- PORT LIST
  port(
    Y : in std_logic_vector(7 downto 0);
    Z : out std_logic_vector(7 downto 0)
  );
end inc8;

-- VHDL object: Architecture "structure" (schematic */home/rjh6v/csis/mentor/B4/eddm/work/
inc8/inc8_gen_meds')of entity "inc8" (component interface * /home/rjh6v/csis/mentor/B4/eddm/
work/inc8*)

```

F.22 DEC4

F.22.1 dec4.vhd

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity dec4 is
port(
    y: in INTEGER range -7 to 7;
    z: out INTEGER range -8 to +6
);
end dec4;

architecture behave of dec4 is
begin
process(y)
begin
    z <= y - 1;
end process;
end behave;

```

F.22.2 dec4_vectors.txt

```

$ y z : time;
0000 1111 : 500 ns;
1111 1110 : 500 ns;
$;
0001 0000 : 500 ns;
0010 0001 : 500 ns;
0100 0011 : 500 ns;
1000 0111 : 500 ns;

```

F.22.3 dec4_structure.vhd

```

-- VHDL object: Entity "dec4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
dec4;dec4")
-- Generated on: Wed May 27 14:10:55 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/dec4/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- PORT LIST
port(
    y : in std_logic_vector(3 downto 0);
    z : out std_logic_vector(3 downto 0)
);
end dec4;

```

```

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
dec4_gen_med_s") of entity "dec4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/
work/dec4;dec4")
-- Generated on: Wed May 27 14:10:55 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/dec4/dec4_gen_med.s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

F.23 ADDINC4**F.23.1 addinc4.vhd**

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;
entity addinc4 is
port(
    OP: in std_logic;
    A: in std_logic_vector(3 downto 0);
    B: in std_logic_vector(3 downto 0);
    D: out std_logic_vector(3 downto 0));
end addinc4;
architecture behave of addinc4 is
begin
process(OP,A,B)
begin
    if OP = '0' then
        D <= A + B;
    else
        D <= A + "0001";
    end if;
end process;
end behave;

```

F.23.2 addinc4_vectors.txt

```

% OP A B D : time :
% ADD
0 0000 0000 0000 : 500 ns;
0 0000 1111 1111 : 500 ns;
0 1111 0000 1111 : 500 ns;
0 1111 1111 1110 : 500 ns;
%
0 0001 1111 0000 : 500 ns;
0 1111 0001 0000 : 500 ns;
0 0101 0101 1010 : 500 ns;
0 1010 1010 0100 : 500 ns;
%
1 INC
0 0000 XXXX 0001 : 500 ns;
1 1111 XXXX 0000 : 500 ns;
%
1 0111 XXXX 1000 : 500 ns;
1 1011 XXXX 1100 : 500 ns;
1 1101 XXXX 1110 : 500 ns;
1 1110 XXXX 1111 : 500 ns;

```

F.23.3 addinc4_structure.vhd

```

-- VHDL object: Entity "addinc4" (component interface )
library ieee;
use ieee.std_logic_1164.all;
entity addinc4 is
    port(
        op : in std_logic;

```

F.24 PLUS3**F.24.1 plus3.vhd**

```
-- VHDL object: Entity 'plus3' (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
plus3:plus3")
-- Generated on: Wed Aug 5 14:25:42 1998
-- Generated by: rjh6v
-- Source from: "/home/rjh6v/csis/mentor/B4/eddm/work/plus3/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

LIBRARY IEEE;
PACKAGE STATEMENT
USE ieee.std_logic_1164.ALL;
entity plus3 is
port(
    y : in std_logic_vector(4 downto 0);
    z : out std_logic_vector(4 downto 0)
);
end plus3;

architecture behave of plus3 is
begin
process(y)
begin
    z <= y + "00011";
end process;
end behave;
```

F.24.2 plus3_vectors.txt

```
% y z : time;
00000 00011 : 500 ns;
11111 00010 : 500 ns;
%
f1100 11111 : 500 ns;
11001 11100 : 500 ns;
10011 10110 : 500 ns;
00111 01010 : 500 ns;
01110 10001 : 500 ns;
```

F.24.3 plus3_structure.vhd

```
-- VHDL object: Entity 'plus3' (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
plus3:plus3")
-- Generated on: Wed Aug 5 14:25:42 1998
-- Generated by: rjh6v
-- Source from: "/home/rjh6v/csis/mentor/B4/eddm/work/plus3/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

LIBRARY IEEE;
PACKAGE STATEMENT
USE ieee.std_logic_1164.ALL;
entity plus3 is
port(
    y : in std_logic_vector(4 downto 0);
    z : out std_logic_vector(4 downto 0)
);
end plus3;

-- VHDL object: Architecture "structure" of entity "plus3" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
plus3:plus3")
-- Generated on: Wed Aug 5 14:25:42 1998
-- Generated by: rjh6v
-- Source from: "/home/rjh6v/csis/mentor/B4/eddm/work/plus3/plus3_low_s"
```

F.25 MINUS5

F.25.1 minus5.vhd

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC std_logic_arith.all;

entity minus5 is
  port(
    Y: in std_logic_vector(5 downto 0);
    Z: out std_logic_vector(5 downto 0)
  );
end minus5;

architecture behave of minus5 is
begin
  process(Y)
  begin
    Z <= Y - "00101";
  end process;
end behave;

```

F.25.2 minus5_vectors.txt

```

% Y Z : time;
00000 111011 : 500 ns;
11111 111010 : 500 ns;
%
000101 000000 : 500 ns;
001010 000101 : 500 ns;
010100 011111 : 500 ns;
101000 100011 : 500 ns;
010001 001100 : 500 ns;
100010 011101 : 500 ns;

```

F.25.3 minus5_structure.vhd

```

-- VHDL object: Entity 'minus5' (component interface) "/home/rjh6v/csis/mentor/B4/eddm/work/
minus5/minus5"
-- Generated on: Wed Aug 5 14:30:28 1998
-- Generated by: rjh6v
-- Source from: "/home/rjh6v/csis/mentor/B4/eddm/work/minus5/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:16:22 PST 1997
-- LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity minus5 is
  -- PORT LIST
  port(
    Y : in std_logic_vector(5 downto 0);
    Z : out std_logic_vector(5 downto 0)
  );
end minus5;

```

```

-- VHDL object: Architecture 'structure' (schematic) "/home/rjh6v/csis/mentor/B4/eddm/work/
minus5/minus5_gen_low_s/minus5"
-- Generated on: Wed Aug 5 14:30:28 1998
-- Generated by: rjh6v

```

F.26 PLUS25**F.26.1 plus25.vhd**

```

library IEEE; ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity plus25 is
  port(
    Y: in INTEGER range -128 to +102;
    Z: out INTEGER range -103 to +127
  );
end plus25;

architecture behave of plus25 is
begin
  process(Y)
  begin
    Z <= Y + 25;
  end process;
end behave;

```

F.26.2 plus25_vectors.txt

```

% Y Z : time;
00000000 00011001 : 500 ns;
11111111 00011000 : 500 ns;
% Y Z : time;
11100110 11111111 : 500 ns;
11001100 11100110 : 500 ns;
10011011 10110010 : 500 ns;
00110111 01010000 : 500 ns;
11011010 10000111 : 500 ns;
11011100 11110101 : 500 ns;
10111001 11010010 : 500 ns;
01110011 10001100 : 500 ns;

```

F.26.3 plus25_structure1.vhd

```

-- VHDL object: Entity "plus25" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
plus25/plus25")
-- Generated on: Wed Jul 29 15:03:17 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/plus25/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
entity plus25 is
  port(
    y : in std_logic_vector(7 downto 0);
    z : out std_logic_vector(7 downto 0)
  );
end plus25;

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
plus25/plus25_gen_low_s") of entity "plus25" (component interface "/home/rjh6v/csis/mentor/B4/
eddm/work/plus25/plus25")

```

F.26.4 plus25_structure2.vhd

```

-- Generated on: Wed Jul 29 15:03:17 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
use ieee.std_logic_1164.ALL;
architecture structure2 of plus25 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
  component and3
  component inv
  component nand3
  component nor2
  component or2
  component xor2
  -- INLINE CONFIGURATIONS
  begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
  end structure2;

```

F.27 MINUS25**F.27.1 minus25.vhd**

```
library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;

entity minus25 is
  port(
    Y: in INTEGER range -103 to +127;
    Z: out INTEGER range -128 to +102
  );
end minus25;

architecture behave of minus25 is
begin
  process(Y)
    begin
      Z <= Y - 25;
    end process;
  end behave;
```

F.27.2 minus25_vectors.txt

```
% Y Z : Time:
00000000 11100111 : 500 ns;
11111111 11100110 : 500 ns;
%
00011001 00000000 : 500 ns;
00110010 00011001 : 500 ns;
01100100 01001011 : 500 ns;
11001000 10101111 : 500 ns;
10010001 01111000 : 500 ns;
00100011 00001010 : 500 ns;
01000110 00101101 : 500 ns;
10001100 01110011 : 500 ns;
```

F.27.3 minus25_structure.vhd

```
-- VHDL object: Entity "minus25" (component interface "home/rjh6v/csis/mentor/B4/eddm/work/
minus25/minus25")
-- Generated on: Mon Aug  3 15:45:27 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/minus25/part
-- Program:  VHDLwrite_v8.5_2.4 Wed Feb 5 16:36:22 PST 1997

-- LIBRARY IEEE;
-- PACKAGE STATEMENT
use IEEE.std_logic_1164.ALL;
entity minus25 is
  port(
    Y : in std_logic_vector(7 downto 0);
    Z : out std_logic_vector(7 downto 0)
  );
end minus25;

-- VHDL object: Architecture "structure" (schematic "home/rjh6v/csis/mentor/B4/eddm/work/
minus25/minus25_gen_low_s") of entity minus25 (componentinterface "/home/rjh6v/csis/mentor/
B4/eddm/work/minus25/minus25")
-- Generated on: Mon Aug  3 15:45:27 1998
-- Generated by: rjh6v
```

F.28 SOP1**F.28.1 sop1.vhd**

```
library IEEE;
library work;
use IEEE.std_logic_1164.all;

entity sop1 is
  port(
    A, B, C, D: in std_logic;
    Z: out std_logic
  );
end sop1;

architecture behave of sop1 is
begin
  process(A,B,C,D)
  begin
    Z <= (A AND B) OR (C AND D);
  end process;
  end behave;
```

F.28.2 sop1_vectors.txt

```
% A B C D Z : time ;
0101 0 : 500 ns;
0X10 0 : 500 ns;
0X11 1 : 500 ns;
100X 0 : 500 ns;
110X 1 : 500 ns;
```

F.28.3 sop1_structure1.vhd

```
-- VHDL object: Entity "sop1" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
sop1:sop1")
-- Generated on: Thu Apr 30 11:34:44 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sop1/sop1.vhd
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity sop1 is
  port(
    a : in std_logic;
    b : in std_logic;
    c : in std_logic;
    d : in std_logic;
    z : out std_logic
  );
end sop1;

-- VHDL object: Architecture "structure" of entity "sop1" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
sop1:sop1")
-- Generated on: Thu Apr 30 11:34:45 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sop1/sop1_auto_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of sop1 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
  component or2
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure;
```

F.28.4 sop1_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "componentInterface") of entity "sop1" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
sop1:sop1")
-- Generated on: Thu Apr 30 11:48:46 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sop1/sop1.vhd
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of sop1 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component nand2
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;
```

F.29 SOP4**F.29.1 sop4.vhd**

```

library ieee;
library work;
use ieee.std_logic_1164.all;
entity sop4 is
port(A, B, C, D: in std_logic_vector(3 downto 0);
      Z: out std_logic_vector(3 downto 0));
end sop4;
architecture behave of sop4 is
begin
process(A,B,C,D)
begin
  Z <= (A AND B) OR (C AND D);
end process;
end behave;

```

F.29.2 sop4_vectors.txt

```

% A B C D Z : time ; 0000 : 500 ns ;
0000 1111 0000 1111 0000 : 500 ns ;
0000 1111 0000 0000 : 500 ns ;
0000 XXXX 1111 1111 1111 : 500 ns ;
1111 0000 0000 XXXX 0000 : 500 ns ;
1111 1111 0000 1111 1111 : 500 ns ;

```

F.29.3 sop4_structure1.vhd

```

-- VHDL object: Entity "sop4" (component interface )
-- Generated on: Tue May 5 14:49:47 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sop4/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:16:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity sop4 is
port(
      a : in std_logic_vector(3 downto 0);
      b : in std_logic_vector(3 downto 0);
      c : in std_logic_vector(3 downto 0);
      d : in std_logic_vector(3 downto 0);
      z : out std_logic_vector(3 downto 0)
);
end sop4;

```

-- VHDL object: Architecture "structure" (schematic)
-- Generated on: Tue May 5 14:49:47 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sop4_auto_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:36:22 PST 1997

```

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure1 of sop4 is
begin
end structure1;

```

F.29.4 sop4_structure2.vhd

```

-- VHDL object: Architecture "structure" (componentInterface )
-- Generated on: Tue May 5 14:58:21 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/sop4_hand_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of sop4 is
begin
end structure2;

```

F.30 POS1**F.30.1 pos1.vhd**

```

library IEEE;
use IEEE.std_logic_1164.all;

entity pos1 is
  port(
    A, B, C, D: in std_logic;
    Z: out std_logic
  );
end pos1;

architecture behave of pos1 is
begin
  process(A,B,C,D)
  begin
    Z <= (A NOR B) NOR (C NOR D);
  end process;
end behave;

```

F.30.2 pos1_vectors.txt

```

% A B C D Z : time ;
001X 0 : 500 ns;
011X 1 : 500 ns;
1010 1 : 500 ns;
1X00 0 : 500 ns;
1X01 1 : 500 ns;

```

F.30.3 pos1_structure1.vhd

```

-- VHDL object: Entity 'pos1' (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
pos1/pos1") 
-- Generated on: Thu Apr 30 12:44:54 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/pos1/pos1
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE IEEE.std_logic_1164.all;
entity pos1 is
  -- PORT LIST
  port(
    a : in std_logic;
    b : in std_logic;
    c : in std_logic;
    d : in std_logic;
    z : out std_logic
  );
end pos1;

```

F.30.4 pos1_structure2.vhd

```

-- VHDL object: Architecture "structure" (schematic "componentInterface"/componentInterface" pos1/pos1/pos1")
-- Generated on: Thu Apr 30 12:51:03 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/pos1/pos1_low_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of pos1 is
begin
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component nor2
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;

```

```

-- VHDL object: Architecture "structure" (schematic "componentInterface"/componentInterface" pos1/pos1/pos1/pos1_auto_s")
-- Generated on: Thu Apr 30 12:44:54 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/pos1/pos1_auto_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

F.31 GT

F.31.1 gt.vhd

```
library IEEE;
use IEEE.std_logic.all;

entity gt is
  port(
    A, B: in std_logic_vector(1 downto 0);
    GT: out std_logic
  );
end gt;
```

architecture behave of gt is

```
begin
  process(A,B)
    begin
      GT <= (A(1) AND not B(1)) OR (A(0) AND not B(0)) AND (A(1) OR not B(1));
    end process;
  end behave;
```

F.31.2 gt_vectors.txt

```
% A B GT : time ;
00 0X 0 : 500 ns;
01 00 1 : 500 ns;
01 X1 0 : 500 ns;
01 10 0 : 500 ns;
10 0X 1 : 500 ns;
10 10 0 : 500 ns;
11 10 1 : 500 ns;
11 11 0 : 500 ns;
```

F.31.3 gt_structure1.vhd

```
-- VHDLObject: Entity "gt" (componentInterface "/home/rjh6v/csis/mentor/B4/eddm/work/gt:gt")
-- Generated on: Thu Apr 30 13:2:16 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/gt/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.ALL;
entity gt is
  port(
    a : in std_logic_vector(1 downto 0);
    b : in std_logic_vector(1 downto 0);
    gt : out std_logic
  );
end gt;
-- VHDLObject: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/gt/gt_low_s")
-- Generated on: Thu Apr 30 13:21:16 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/gt:auto_S
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.ALL;
```

F.32 XOR4

F.32.1 xor4.vhd

```
Library ieee;
use ieee.std_logic_1164.all;
entity XOR4 is
port(
    A, B, C, D : in std_logic;
    Z : out std_logic
);
end XOR4;

architecture behave of XOR4 is
begin
    Z <= A XOR B XOR C XOR D;
end behave;
```

F.32.2 xor4_vectors.txt

```
% ABCD Z : time ;
0000 0 : 500 ns;
0001 1 : 500 ns;
0010 1 : 500 ns;
0011 0 : 500 ns;
0100 1 : 500 ns;
0110 0 : 500 ns;
1000 1 : 500 ns;
1100 0 : 500 ns;
```

F.32.3 xor4_vectors.opt

```
% ABCD Z : time ;
0000 0 : 500 ns;
0111 1 : 500 ns;
1001 0 : 500 ns;
1110 1 : 500 ns;
```

F.32.5 xor4_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic *)/mentor/B4/eddm/work/
xor4/xor4
-- Generated on: Fri May 8 10:35:14 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/xor4/xor4_auto_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
use ieee.std_logic_1164.all;
architecture structure2 of xor4 is
begin
    Z <= A XOR B XOR C XOR D;
end structure2;

-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
library work;
use ieee.std_logic_1164.all;
component xor2
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        d : in std_logic;
        z : out std_logic
    );
end component xor2;
begin
    Z <= A XOR B XOR C XOR D;
end structure2;
```

F.32.4 xor4_structure1.vhd

```
-- VHDL object: Entity "xor4" (component interface *)/mentor/B4/eddm/work/
xor4/xor4
-- Generated on: Fri May 8 10:35:14 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/xor4/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
library ieee;
use ieee.std_logic_1164.all;
entity xor4 is
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        d : in std_logic;
        z : out std_logic
    );
end entity xor4;
```

F.32.6 xor4_structure3.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/xor4/xor4_xi72a_s") of entity "xor4"
-- Generated on: Tue Jul 21 11:16:32 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/xor4/xor4_xi72a_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of xor4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
    component inv
    component or2
  end component;
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure;
```

F.32.7 xor4_structure4.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/schem/xor4_nand/schematic") of entity "xor4_nand"
-- Generated on: Fri May 8 12:24:46 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/schem/xor4_nand/schematic
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure4 of xor4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component nand2
  end component;
  -- INLINE CONFIGURATIONS
begin
  -- COMPONENT INSTANTIATIONS
end structure4;
```

```

F.33 XOR5
F.33.1 xor5.vhd
Library ieee;
use ieee.std_logic_1164.all;
entity XOR5 is
port(A, B, C, D, E : in std_logic;
     Z : out std_logic
);
end XOR5;

architecture behave of XOR5 is
begin
  Z <= A XOR B XOR C XOR D XOR E;
end behave;

F.33.2 xor5_vectors.txt
% ABCDE Z : time ;
00000 0 : 500 ns;
10110 1 : 500 ns;
01011 1 : 500 ns;
11101 0 : 500 ns;

F.33.3 xor5_structure1.vhd
-- VHDL object: Entity "xor5" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/xor5")
-- Generated on: Fri May 8 11:04:14 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/xor5/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
library ieee;
-- PACKAGE STATEMENT
use ieee.std_logic_1164.all;
entity xor5 is
  port(
    a : in std_logic;
    b : in std_logic;
    c : in std_logic;
    d : in std_logic;
    e : in std_logic;
    z : out std_logic
  );
end xor5;

F.33.4 xor5_structure2.vhd
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/xor5/xor5_c2_s") of entity "xor5" (componentinterface "/home/rjh6v/csis/mentor/B4/eddm/work/xor5")
-- Generated on: Fri May 8 11:13:10 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/xor5/xor5_c2_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of xor5 is
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure1;

F.33.5 xor5_structure3.vhd
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/xor5_nand") of entity "xor5_nand" (componentinterface "/home/rjh6v/csis/mentor/B4/eddm/work/xor5_nand")
-- Generated on: Fri May 8 12:42:25 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/schem/xor5_nand/schematic
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure3 of xor5 is
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
  -- INLINE CONFIGURATIONS
end structure2;

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/xor5_auto_s") of entity "xor5_auto_s" (componentinterface "/home/rjh6v/csis/mentor/B4/eddm/work/xor5")
-- Generated on: Fri May 8 11:04:14 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/xor5/xor5_auto_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;

```

F.33.6 xor5_structure4.vhd

```
--VHDLobject:Architecture"structure"(schematic"/home/rjh6v/csis/mentor/B4/schem/xor5_b2/
schematic") of entity "xor5_b2" (component interface "/home/rjh6v/csis/mentor/B4/schem/
xor5_b2;xor5_b2")
-- Generated on: Fri May 22 09:47:19 1998
-- Generated by: rjnv
-- Source from: /home/rjh6v/csis/mentor/B4/schem/xor5_b2/schematic
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure4 of xor5 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component nand2
    -- INLINE CONFIGURATIONS
  begin
    -- COMPONENT INSTANTIATIONS
  end structure4;
```

F.34 ABS4**F.34.1 abs4.vhd**

```
library ieee;
use ieee.std_logic_1164.all;
entity ABS4 is
port(X: in INTEGER range -7 to +7;
     Z: out INTEGER range 0 to +7);
end ABS4;
```

architecture BEHAVE of ABS4 is

```
begin
process(X)
begin
    Z <= abs X;
end process;
end BEHAVE;
```

F.34.2 abs4_vectors.txt

```
% X      Z      : time ;
0000 000 : 500 ns;
0111 111 : 500 ns;
1001 111 : 500 ns;
1010 110 : 500 ns;
1100 100 : 500 ns;
```

F.34.3 abs4_structure1.vhd

```
-- VHDL object: Entity "abs4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4")
-- Generated on: Wed May 13 14:59:21 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs4/part
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
entity abs4 is
port(x : in std_logic_vector(3 downto 0);
     Z : out std_logic_vector(12 downto 0));
end abs4;
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4/abs4_low_s/low_s/abs4")
-- Generated on: Wed May 13 14:59:21 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs4/abs4_low_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
library work;
use ieee.std_logic_1164.ALL;
architecture structure of abs4 is
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
-- component_and2
-- component_buf
-- component_inv
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4/abs4_low_s/low_s/abs4")
-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
end structure;
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4/abs4_low_s/low_s/abs4")
-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
begin
    -- concurrent signal assignments
    -- component declarations
    -- component instantiations
end structure;
```

```
-- VHDL object: Entity "abs4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4")
-- Generated on: Tue May 12 14:41:34 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs4/abs4_low_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
component nand2p
component or2
component xor2
```

```
-- INLINE CONFIGURATIONS
```

```
begin
    -- concurrent signal assignments
    -- component declarations
    end structure;
```

F.34.4 abs4_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4/abs4_low_s/low_s/abs4")
-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
begin
    -- concurrent signal assignments
    -- component declarations
    -- component instantiations
end structure;
```

```
-- VHDL object: Entity "abs4" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4")
-- Generated on: Tue May 12 14:41:34 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs4/abs4_low_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs4")
-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
begin
    -- concurrent signal assignments
    -- component declarations
    -- component instantiations
end structure;
```

F.35 ABS8**F.35.1 abs8.vhd**

```

library ieee;
library work;
use ieee.std_logic_1164.all;
entity ABS8 is
port(X: in INTEGER range -127 to +127;
      Z: out INTEGER range 0 to +127);
end ABS8;

architecture BEHAVE of ABS8 is
begin
process(X)
begin
  Z <= abs X;
end process;
end BEHAVE;

```

F.35.2 abs8_vectors.txt

```

% X          Z          : time
00000000 00000000 : 500 ns;
01111111 11111111 : 500 ns;
10000001 11111111 : 500 ns;
10000010 11111110 : 500 ns;
10000100 11111000 : 500 ns;
10001000 11100000 : 500 ns;
10010000 11000000 : 500 ns;
11000000 10000000 : 500 ns;

```

F.35.3 abs8_structure1.vhd

```

-- VHDL object: Entity "abs8" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
abs8;abs8")
-- Generated on: Wed May 13 16:27:41 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs8/part
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.ALL;
entity abs8 is
-- PORT LIST
port(
      X : in std_logic_vector(7 downto 0);
      Z : out std_logic_vector(6 downto 0)
);
end abs8;

```

```

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs8;abs8_low_s") of entity "abs8"
-- Generated on: Wed May 13 16:27:41 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs8_low_s
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

```

F.35.4 abs8_structure2.vhd

```

library ieee;
library work;
use ieee.std_logic_1164.ALL;
architecture structure of abs8 is
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
abs8;abs8_low_s") of entity "abs8" (component interface "/home/rjh6v/csis/mentor/B4/eddm/
work/abs8;abs8_low_s")
-- Generated on: Wed May 13 16:29:07 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/abs8_low_s
-- Program: VHDWrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- CONCURRENT SIGNAL ASSIGNMENTS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
  end structure;

```

F.36 NEG4**F.36.1 neg4.vhd**

```
library ieee;
use ieee.std_logic_1164.all;
entity NEG4 is
    port(X: in INTEGER range -7 to +7;
         Z: out INTEGER range -7 to +7);
end NEG4;
```

architecture BEHAVE of NEG4 is

```
begin
    process(X)
    begin
        Z <= -X;
    end process;
end BEHAVE;
```

F.36.2 neg4_vectors.txt

```
% X      : time ;
0000 0000 : 500 ns;
0111 1001 : 500 ns;
1001 0111 : 500 ns;
1010 0110 : 500 ns;
1100 0100 : 500 ns;
```

F.36.3 neg4_structure1.vhd

```
-- VHDL object: Entity 'neg4' (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/neg4")
-- Generated on: Wed May 13 15:04:38 1998
-- Generated by: rjh6v
-- Source From: /home/rjh6v/csis/mentor/B4/eddm/work/neg4/part
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

entity neg4 is

-- PORT LIST

```
port
    X : in std_logic_vector(3 downto 0);
    Z : out std_logic_vector(3 downto 0);
);
```

end neg4;

```
-- VHDL object: Architecture 'structure' (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/neg4/neg_low_s") of entity 'neg4' (componentinterface "/home/rjh6v/csis/mentor/B4/eddm/work/neg4")
-- Generated on: Wed May 13 15:04:38 1998
-- Generated by: rjh6v
-- Source From: /home/rjh6v/csis/mentor/B4/eddm/work/neg4/neg_low_s
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
LIBRARY ieee;
```

library work;

use ieee.std_logic_1164.ALL;

architecture structure1 of neg4 is

-- SIGNAL DECLARATIONS

-- COMPONENT DECLARATIONS

component buf

F.36.4 neg4_structure2.vhd

```
-- VHDL object: Architecture 'structure' (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/neg4/neg_low_s") of entity 'neg4' (component interface "/home/rjh6v/csis/mentor/B4/eddm/")
-- Generated on: Tue May 12 14:52:03 1998
-- Generated by: rjh6v
-- Source From: /home/rjh6v/csis/mentor/B4/eddm/work/neg4/neg_low_s
-- Program: VHDWrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
library work;
use ieee.std_logic_1164.ALL;
architecture structure2 of neg4 is
    -- SIGNAL DECLARATIONS
    -- COMPONENT DECLARATIONS
begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
end structure2;
```

library ieee;

library work;

use ieee.std_logic_1164.ALL;

architecture structure2 of neg4 is
 -- SIGNAL DECLARATIONS
 -- COMPONENT DECLARATIONS
begin
 -- CONCURRENT SIGNAL ASSIGNMENTS
 -- COMPONENT INSTANTIATIONS
end structure2;

F.37 NEG8**F.37.1 neg8.vhd**

```
library ieee;
use ieee.std_logic_1164.all;
entity NEG8 is
port (X: in INTEGER range -127 to +127;
      Z: out INTEGER range -127 to +127);
end NEG8;
```

architecture BEHAVE of NEG8 is

```
begin
```

```
begin
```

```
    Z <= -X;
```

```
end process;
```

```
end BEHAVE;
```

F.37.2 neg8_vectors.txt

```
$ X Z : time ;
00000000 00000000 : 500 ns;
01111111 10000001 : 500 ns;
10000001 01111111 : 500 ns;
10000000 01111110 : 500 ns;
10000100 01111100 : 500 ns;
10010000 01110000 : 500 ns;
10100000 01100000 : 500 ns;
11000000 01000000 : 500 ns;
```

F.37.3 neg8_structure1.vhd

```
-- VHDL object: Entity "neg8" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/neg8")
-- Generated on: Wed May 13 16:50:45 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/neg8/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
```

```
-- LIBRARY statement
```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
```

```
entity neg8 is
```

```
  -- PORT LIST
```

```
  port (
    X : in std_logic_vector(7 downto 0);
    Z : out std_logic_vector(7 downto 0) );
```

```
end neg8;
```

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/neg8_gen_low_s") of entity "neg8" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/neg8_gen_low_s")
-- Generated on: Wed May 13 16:50:45 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/neg8_gen_low_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
```

```
-- LIBRARY statement
```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
```

```
architecture structure1 of neg8 is
```

```
  -- SIGNAL DECLARATIONS
```

```
  -- COMPONENT DECLARATIONS
```

```
  component buf
```

```
    component or2
```

```
    component xor2
```

```
    -- INLINE CONFIGURATIONS
```

```
begin
```

```
  -- CONCURRENT SIGNAL ASSIGNMENTS
```

```
  -- COMPONENT INSTANTIATIONS
```

```
  end structure1;
```

F.37.4 neg8_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/neg8_xi72a_s") of entity "neg8" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/neg8")
-- Generated on: Fri Jul 24 13:35:16 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/neg8_xi72a_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
```

```
library ieee;
```

```
library work;
```

```
use ieee.std_logic_1164.ALL;
```

```
architecture structure2 of neg8 is
```

```
  -- SIGNAL DECLARATIONS
```

```
  -- COMPONENT DECLARATIONS
```

```
  component and2
```

```
  component inv
```

```
  component buf
```

```
  component or2
```

```
  -- INLINE CONFIGURATIONS
```

```
begin
```

```
  -- CONCURRENT SIGNAL ASSIGNMENTS
```

```
  -- COMPONENT INSTANTIATIONS
```

```
  end structure2;
```

F.38 ALU4wc

F.38.1 alu4wc.vhd

```

library IEEE, ARITHMETIC;
use IEEE.std_logic_1164.all;
use ARITHMETIC.std_logic_arith.all;
-- Written by LL-to-VHDL at Mon Jun  8 12:23:29 1998
-- Parameterized Generator Specification to VHDL Code
-- LogicLib generator called: ARITHMETIC
-- Passed Parameters are:
--   inst_name = alu4wc
--   parameters are:
--     type = ALU2901
--     W = 4
--     carryin = YES
--     carryout = YES
--   alu4wc Entity Description
entity alu4wc is
begin
  -- alu4wc Architecture Description
  architecture behave of alu4wc is
    begin
      -- zero extend inputs to include carry bit
      a_ext := '0' & A;
      b_ext := '0' & B;
      not_a_ext := '0' & not A;
      not_b_ext := '0' & not B;
      carry_ext := '0' & CIN;
      -- ALU2901
      -- Logical Functions --
      case OP is
        when "011" =>
          logic_out := A or B;
        when "100" =>
          logic_out := A and B;
        when "101" =>
          logic_out := (not A) and B;
        when "110" =>
          logic_out := A xor B;
        when "111" =>
          logic_out := not (A xor B);
        when others =>
          logic_out := (OTHERS => 'X');
      end case;
    end;
  end;
end;

```

F.38.2 alu4wc_functions.txt

```

ARITHMETIC_Process: process(A,B,CIN,OP)
variable operand1, operand2 : std_logic_vector(4 downto 0);
variable a_ext, b_ext : std_logic_vector(4 downto 0);
variable not_a_ext, not_b_ext : std_logic_vector(4 downto 0);
variable carry_ext : std_logic_vector(1 downto 0);
variable logic_out : std_logic_vector(3 downto 0);
variable arith_out : std_logic_vector(3 downto 0);
begin
  -- zero extend inputs to include carry bit
  a_ext := '0' & A;
  b_ext := '0' & B;
  not_a_ext := '0' & not A;
  not_b_ext := '0' & not B;
  carry_ext := '0' & CIN;
  -- ALU2901
  -- Logical Functions --
  case OP is
    when "011" =>
      logic_out := A or B;
    when "100" =>
      logic_out := A and B;
    when "101" =>
      logic_out := (not A) and B;
    when "110" =>
      logic_out := A xor B;
    when "111" =>
      logic_out := not (A xor B);
    when others =>
      logic_out := (OTHERS => 'X');
  end case;
end;

```

F.38.3 alu4wc_vectors.txt

```

% OP  A  B  CIN  COUT  D : time;
% 000 0000 0 00000 : 500 ns;
% 000 1111 1111 1 1111 : 500 ns;
% 000 0101 0101 0 1010 : 500 ns;
% 000 1010 1010 1 0101 : 500 ns;
% 001 1111 0000 0 00000 : 500 ns;
% 001 0000 0000 0 01111 : 500 ns;
% 001 1010 0101 0 1010 : 500 ns;
% 001 0101 1010 1 0101 : 500 ns;
% 010 0000 1111 0 0 00000 : 500 ns;
% 010 0000 0000 0 11111 : 500 ns;
% 010 0000 0000 1 1 00000 : 500 ns;
% 010 1111 1111 1 1 00000 : 500 ns;
% 011 0000 0000 0 X 00000 : 500 ns;
% 011 0000 1111 0 X 1111 : 500 ns;
% 011 1111 0000 0 X 1111 : 500 ns;
% 100 0000 1111 0 X 00000 : 500 ns;
% 100 1111 0000 X 00000 : 500 ns;
% 100 0101 0101 0 X 1010 : 500 ns;
% 100 1010 1010 1 X 1010 : 500 ns;
% 101 1111 1111 0 X 00000 : 500 ns;
% 101 0000 0000 X 00000 : 500 ns;
% 101 1010 0101 0 X 1010 : 500 ns;
% 101 0101 1010 1 X 1010 : 500 ns;
%
```

```

-- CONCURRENT SIGNAL ASSIGNMENTS
-- COMPONENT INSTANTIATIONS
end structure1;

F.38.4 alu4wc_structure2.vhd

-- VHDL object: Architecture "structure" (schematic "home/rjh6v/csis/mentor/B4/eddm/work/
alu4wc/alu4wc_high_s") of entity alu4wc (componentinterface /home/rjh6v/csis/mentor/B4/eddm/
work/alu4wc/alu4wc");
-- Generated on: Tue Jun 16 11:18:16 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/alu4wc/alu4wc_high_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of alu4wc is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
begin
  component and2
  component and3
  component invr
  component nandp
  component nand3p
  component nand4p
  component nor2p
  component or2
  component or2p
  component xor2
  end component;
  -- INLINE CONFIGURATIONS
  begin
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
    end structure2;
  end;

F.38.3 alu4wc_structure1.vhd

-- VHDL object: Entity "alu4wc" (component interface "/home/rjh6v/csis/mentor/B4/eddm/work/
alu4wc:alu4wc")
-- Generated on: Thu Jun 11 16:01:32 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/alu4wc/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:32 PST 1997

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity alu4wc is
  -- PORT LIST
  port(
    op : in std_logic_vector(2 downto 0);
    a : in std_logic_vector(3 downto 0);
    b : in std_logic_vector(3 downto 0);
    cin : in std_logic;
    cout : out std_logic;
    d : out std_logic_vector(3 downto 0)
  );
end alu4wc;

-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
alu4wc/alu4wc_gen_high_s") of entity alu4wc (componentinterface /home/zjh6v/csis/mentor/B4/
eddm/work/alu4wc_gen_high_s");
-- Generated on: Tue Jun 16 14:55:51 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/alu4wc/alu4wc_gen_high_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure1 of alu4wc is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
begin
  component and2
  component and3
  component invr
  component nand2
  component nand3
  component nor2
  component nor3
  component or2
  component or4
  component xor2
  end component;
  -- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure1;

```

F.39 ALU8wc**F.39.1 alu8wc.vhd**

```

library IEEE, ARITHMETIC;
use STD_logic.all;
use ARITHMETIC.std_logic_arith.all;

-- Written by LU_to_VHDL at Mon Jun  8 12:23:29 1998
-- Parameterized Generator Specification to VHDL Code
-- LogicLib generator called: ARITHMETIC
-- Passed Parameters are:
--   inst name = alu8wc
--   parameters are:
--     type = ALU2901
--     W = 8
--     carryin = YES
--     carryout = YES

-- alu8wc Entity Description
entity alu8wc is
port(
    OP: in std_logic_vector(2 downto 0);
    A: in std_logic_vector(7 downto 0);
    B: in std_logic_vector(7 downto 0);
    CIN: in std_logic;
    COUT: out std_logic;
    D: out std_logic_vector(7 downto 0)
);
end alu8wc;

-- alu8wc Architecture Description
architecture behave of alu8wc is
begin
    ARITHMETIC_Process: process(A,B,CIN,OP)
        variable operand1, operand2 : std_logic_vector(8 downto 0);
        variable a_ext, b_ext : std_logic_vector(8 downto 0);
        variable not_a_ext, not_b_ext : std_logic_vector(8 downto 0);
        variable carry_ext : std_logic_vector(1 downto 0);
        variable logic_out : std_logic_vector(7 downto 0);
        variable arith_out : std_logic_vector(8 downto 0);
    begin
        -- zero extend inputs to include carry bit
        a_ext := '0' & A;
        b_ext := '0' & B;
        not_a_ext := '0' & not A;
        not_b_ext := '0' & not B;
        carry_ext := '0' & CIN;
    end;
    -- ALU2901
    -- -----
    -- Logical Functions --
    case OP is
        when "011" =>
            logic_out := A or B;
        when "100" =>
            logic_out := A and B;
        when "101" =>
            logic_out := (not A) and B;
        when "110" =>
            logic_out := A xor B;
        when "111" =>
    end case;
end;

```

```

        logic_out := not (A xor B);
        when others =>
            logic_out := (others => 'X');
    end case;
    -- Arithmetic Functions --
    case OP is
        -- Arithmetic operations
        when "000" =>
            operand1 := a_ext;
            operand2 := b_ext;
        when "001" =>
            operand1 := not_a_ext;
            operand2 := b_ext;
        when "010" =>
            operand1 := a_ext;
            operand2 := not_b_ext;
        when others =>
            operand1 := (others => 'X');
            operand2 := (others => 'X');
    end case;
    arith_out := operand1 + operand2 + carry_ext;
    -- assign output
    if (OP(2) = '1' or (OP(1) = '1' and OP(0) = '1')) then
        D <= logic_out;
        COUT <= 'X';
    else
        D <= arith_out(7 downto 0);
        COUT <= arith_out(8);
    end if;
end process ARITHMETIC_Process;
end behave;

```

F.39.2 alu8wc_vectors.txt

```

% OP A B CIN COUNT D : time;
000 00000000 00000000 0 0 00000000 : 500 ns;
000 11111111 11111111 1 1 11111111 : 500 ns;
000 01010101 01010101 0 0 10101010 : 500 ns;
000 10101010 10101010 1 1 01010101 : 500 ns;
%
001 11111111 00000000 0 0 00000000 : 500 ns;
001 00000000 00000000 0 0 11111111 : 500 ns;
001 10101010 10101010 0 0 10101010 : 500 ns;
001 01010101 10101010 1 1 01010101 : 500 ns;
%
010 00000000 11111111 0 0 00000000 : 500 ns;
010 00000000 00000000 0 0 11111111 : 500 ns;
010 00000000 00000000 1 1 00000000 : 500 ns;
010 11111111 11111111 1 1 00000000 : 500 ns;
%
011 00000000 00000000 0 X 00000000 : 500 ns;
011 00000000 11111111 0 X 11111111 : 500 ns;
011 11111111 00000000 0 X 11111111 : 500 ns;
%
100 00000000 11111111 0 X 00000000 : 500 ns;
100 11111111 00000000 X X 00000000 : 500 ns;
100 01010101 01010101 0 X 10101010 : 500 ns;
100 10101010 10101010 1 X 10101010 : 500 ns;

```

```


-- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure1;

F.39.4 alu8wc_structure2.vhd
-----

-- VHDL object: Architecture "structure2" (schematic "home/rjh6v/csis/mentor/B4/eddm/work/
alu8wc/alu8wc_xi72a_s") of entity "alu8wc" (component interface /home/rjh6v/csis/mentor/B4/
eddm/work/alu8wc:alu8wc")
-- Generated on: Wed Jun 24 12:06:27 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/alu8wc/alu8wc_xi72a_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:36:22 PST 1997
-----
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of alu8wc is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component or2
    end component;
  begin
    -- INLINE CONFIGURATIONS
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
    end structure2;

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure1 of alu8wc is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component or2
    end component;
  begin
    -- LIBRARY STATEMENT
    LIBRARY ieee;
    -- PACKAGE STATEMENT
    USE ieee.std_logic_1164.all;
    entity alu8wc is
      -- PORT LIST
      port(
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        cin : in std_logic;
        op : in std_logic_vector(2 downto 0);
        cout : out std_logic;
        d : out std_logic_vector(7 downto 0)
      );
    end alu8wc;
  end structure1;

F.39.3 alu8wc_structure1.vhd
-----

-- VHDL object: Entity "alu8wc" (component interface /home/rjh6v/csis/mentor/B4/eddm/work/
alu8wc:alu8wc)
-- Generated on: Tue Sep 8 11:23:40 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/alu8wc/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:36:22 PST 1997
-----

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity alu8wc is
  -- PORT LIST
  port(
    a : in std_logic_vector(7 downto 0);
    b : in std_logic_vector(7 downto 0);
    cin : in std_logic;
    op : in std_logic_vector(2 downto 0);
    cout : out std_logic;
    d : out std_logic_vector(7 downto 0)
  );
end alu8wc;
-----

-- VHDL object: Architecture "structure1" (schematic "home/rjh6v/csis/mentor/B4/eddm/work/
alu8wc/alu8wc_gen_high_s") of entity "alu8wc" (component interface /home/rjh6v/csis/mentor/B4/
eddm/alu8wc:alu8wc_gen_high_s)
-- Generated on: Tue Sep 8 11:23:40 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/alu8wc/alu8wc_gen_high_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:36:22 PST 1997
-----
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure1 of alu8wc is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component nand2
    component and2
    component inv
    component nor2
    component or2
    component xor2
  end component;
  begin
    -- LIBRARY STATEMENT
    LIBRARY ieee;
    -- PACKAGE STATEMENT
    USE ieee.std_logic_1164.all;
    entity alu8wc is
      -- PORT LIST
      port(
        a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(7 downto 0);
        cin : in std_logic;
        op : in std_logic_vector(2 downto 0);
        cout : out std_logic;
        d : out std_logic_vector(7 downto 0)
      );
    end alu8wc;
  end structure1;

```

F.40 HAMMING4

F.40.1 hamming4.vhd

```

library IEEE;
use IEEE.std_logic_1164.all;

entity hamming4 is
port(
    X: in std_logic_vector(1 to 4);
    P: in std_logic_vector(1 to 3);
    D: out std_logic_vector(1 to 4)
);
end hamming4;

```

architecture behave of hamming4 is

begin

```

process(X,P)
variable S : std_logic_vector(1 to 3);
begin

```

```

    S(1) := X(1) XOR X(2) XOR X(4) XOR P(1);
    S(2) := X(1) XOR X(3) XOR X(4) XOR P(2);
    S(3) := X(2) XOR X(3) XOR X(4) XOR P(3);
    D(1) <= X(1) XOR (S(1) AND S(2) AND not S(3));
    D(2) <= X(2) XOR (S(1) AND not S(2) AND S(3));
    D(3) <= X(3) XOR (not S(1) AND S(2) AND S(3));
    D(4) <= X(4) XOR (S(1) AND S(2) AND S(3));
end process;

```

end behave;

F.40.2 hamming4_vectors.txt

```

% X P D : time;
0000 000 0000 : 500 ns;
0000 111 0001 : 500 ns;
0001 000 0000 : 500 ns;
0010 101 1010 : 500 ns;
0010 110 0110 : 500 ns;
0010 111 0010 : 500 ns;
0011 101 0011 : 500 ns;
0100 101 0100 : 500 ns;
0101 000 0101 : 500 ns;
0110 011 0010 : 500 ns;
0111 110 0110 : 500 ns;
1000 110 1000 : 500 ns;
1001 000 1001 : 500 ns;
1010 011 0010 : 500 ns;
1100 000 1110 : 500 ns;
1110 011 1100 : 500 ns;
1111 000 1110 : 500 ns;
%
```

F.40.3 hamming4_structure1.vhd

```

-- VHDL object: Entity "hamming4" (component interface
hamming4;
-- Generated on: Thu Jun 18 14:42:04 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming4/part

```

-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

```

-- LIBRARY STATEMENT
LIBRARY ieee;
-- PACKAGE STATEMENT
USE ieee.std_logic_1164.all;
entity hamming4 is
-- PORT LIST
port(
    X : in std_logic_vector(1 to 4);
    P : in std_logic_vector(1 to 3);
    D : out std_logic_vector(1 to 4)
);
end hamming4;

```

```

-- VHDL object: Architecture "structure" (schematic
hamming4_gen_low_s) of entity "hamming4" (component interface
mentor/B4/eddm/work/hamming4);
-- Generated on: Thu Jun 18 14:42:04 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming4_gen_low_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of hamming4 is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component and2
    component inv
        component nand2
            component nor2
                component xor2
                    begin
                        -- CONCURRENT SIGNAL ASSIGNMENTS
                        -- COMPONENT INSTANTIATIONS
                        end structure;

```

F.40.4 hamming4_structure2.vhd

```

-- VHDL object: Architecture "structure" (schematic
hamming4_gen_low_s" of entity "hamming4" (component interface
mentor/B4/eddm/work/hamming4);
-- Generated on: Thu Jun 23 11:38:29 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming4_gen_low_s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of hamming4 is
-- TYPE DECLARATIONS
-- COMPONENT DECLARATIONS
component invr
    component nand2p
        component nand3p
            component nor2p
                component or3p

```

```
component xor2;
component xor3;
-- INLINE CONFIGURATIONS
begin
  -- CONCURRENT SIGNAL ASSIGNMENTS
  -- COMPONENT INSTANTIATIONS
end structure2;
```

F.40.5 hamming4_structure3.vhd

```
-- VPI object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/
hamming4/hamming4_xi72a.s") of entity "hamming4" (component "inteface")
B4 /eddm/work/hamming4/hamming4_xi72a.s
-- Generated on: Tue Jun 23 12:09:05 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming4/hamming4_xi72a.s
-- Program: VHDLwrite v8.5.2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure3 of hamming4 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component inv
    component or2;
    -- INLINE CONFIGURATIONS
    -- CONCURRENT SIGNAL ASSIGNMENTS
    -- COMPONENT INSTANTIATIONS
  end structure3;
```

F.41 HAMMING8

F.41.1 Hamming8.vhd

```

library IEEE;
use IEEE.std_logic_1164.all;

entity hamming8 is
port(
    X: in std_logic_vector(1 to 8);
    P: in std_logic_vector(1 to 4);
    D: out std_logic_vector(1 to 8)
);
end hamming8;

architecture behave of hamming8 is

process(X,P)
variable S : std_logic_vector(1 to 4);
begin
    S(1) := X(1) XOR X(2) XOR X(4) XOR X(5) XOR P(1);
    S(2) := X(1) XOR X(3) XOR X(6) XOR X(7) XOR P(2);
    S(3) := X(2) XOR X(3) XOR X(4) XOR X(8) XOR P(3);
    S(4) := X(5) XOR X(6) XOR X(7) XOR X(8) XOR P(4);

    D(1) <= X(1) XOR (S(1) AND S(2) AND not S(3) AND not S(4));
    D(2) <= X(2) XOR (S(1) AND not S(2) AND S(3) AND not S(4));
    D(3) <= X(3) XOR (not S(1) AND S(2) AND S(3) AND not S(4));
    D(4) <= X(4) XOR (S(1) AND S(2) AND S(3) AND not S(4));
    D(5) <= X(5) XOR (not S(1) AND not S(2) AND S(3) AND S(4));
    D(6) <= X(6) XOR (not S(1) AND S(2) AND not S(3) AND S(4));
    D(7) <= X(7) XOR (S(1) AND S(2) AND not S(3) AND S(4));
    D(8) <= X(8) XOR (not S(1) AND not S(2) AND S(3) AND S(4));

    end process;
end behave;

```

F.41.2 hamming8_vectors.txt

```

X          P          D          time;
00000000 0000 00000000 : 500 ns;
01101111 0000 01111111 : 500 ns;
01110101 0010 01110000 : 500 ns;
10011101 1100 10011101 : 500 ns;
11011101 1101 01101110 : 500 ns;
11101001 1110 11100000 : 500 ns;
00100101 0011 00100100 : 500 ns;
00101011 0001 01011011 : 500 ns;
01011011 0011 01011010 : 500 ns;
01101100 0100 11011100 : 500 ns;
11011100 0001 11011000 : 500 ns;
11001010 0100 11101010 : 500 ns;
11101010 0101 11001010 : 500 ns;
11011010 1011 11011010 : 500 ns;
00100101 1000 00100101 : 500 ns;

```

```

-- VHDL object: Entity "hamming8" (component interface) "/home/rjh6v/csis/mentor/B4/eddm/work/
hamming8;hamming8"
-- Generated on: Fri Jun 26 10:35:55 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming8/part
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
-- LIBRARY STATEMENT
library ieee;
-- PACKAGE STATEMENT
use ieee.std_logic_1164.all;
entity hamming8 is
-- PORT LISTS
port(
    X : in std_logic_vector(1 to 8);
    P : in std_logic_vector(1 to 4);
    D : out std_logic_vector(1 to 8)
);
end hamming8;

-- VHDL object: Architecture "structure" (schematic) "/home/rjh6v/csis/mentor/B4/eddm/work/
hamming8;gen_low_s" of entity "hamming8" (component interface) "/home/rjh6v/csis/
mentor/B4/eddm/work/hamming8"
-- Generated on: Fri Jun 26 10:35:55 1998
-- Generated by: rjh6v
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming8;hamming8_gen_low_s
-- Program: VHDLwrite v8.5_2.4 Wed Feb 5 16:56:22 PST 1997
library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure of hamming8 is
-- TYPE DECLARATIONS
-- SIGNAL DECLARATIONS
-- COMPONENT DECLARATIONS
component nor4
    component inv
        component nor2
            component xor2
                -- INLINE CONFIGURATIONS
                begin
                    CONCURRENT SIGNAL ASSIGNMENTS
                end structure1;
            end component;
        end component;
    end component;
end component;

```

F41.4 Hamming8_structure2.vhd

```
-- VHDL object: Architecture "structure" (schematic "/home/rjh6v/csis/mentor/B4/eddm/work/hamming8/hamming8_xi72a_s") of entity hamming8 ("componentInterface" "/home/rjh6v/csis/mentor/B4/eddm/work/hamming8/hamming8")
-- Generated on: Fri Jun 26 10:59:01 1998
-- Generated by: rjhav
-- Source from: /home/rjh6v/csis/mentor/B4/eddm/work/hamming8/hamming8_xi72a_s
-- Program: VHDWriter v8.5.2.4 Wed Feb 5 16:56:22 PST 1997

library ieee;
library work;
use ieee.std_logic_1164.all;
architecture structure2 of hamming8 is
  -- TYPE DECLARATIONS
  -- SIGNAL DECLARATIONS
  -- COMPONENT DECLARATIONS
  component and2
    component inv
      component or2
        -- INLINE CONFIGURATIONS
        begin
          -- CONCURRENT SIGNAL ASSIGNMENTS
          -- COMPONENT INSTANTIATIONS
      end structure2;
```